DELFT UNIVERSITY OF TECHNOLOGY

MASTER THESIS

CM GROEN 4442806

Grammatical Evolution for Optimising Drone Behaviors

Graduation Date: 11/01/2022

Delft
University of
Technology

Grammatical Evolution for Optimising Drone Behaviors

Christopher Groen, Shushuai Li, Guido de Croon

Abstract— **This paper reviews the application of grammatical evolution for the optimisation of low level parameters and high level behaviors for two drone behaviors, namely wall-following and navigation. In order to optimise these low level parameters and high level behaviors, grammatical evolution was applied to behavior trees. Grammatical evolution provided a significant improvement in the wall-following behavior of a drone, creating a more robust behavior. There was no improvement for the navigation behavior however, with the success rate of navigating deteriorating in some cases. The evolved wall-following behavior was compared and tested against another wall-following controller from literature, and shown to be superior. A real-life experiment was also conducted for the wall-following behavior, which led to positive results after correcting for the reality gap. For the wall-following behavior, the grammatical evolution promoted a continuous scanning behavior, which greatly increased it's awareness of obstacles. Significant recommendations were given to improve the results of the grammatical evolution for both behaviors.**

I. INTRODUCTION

Unmanned aerial vehicles, more commonly known as drones, have been studied rigorously over the past century. Drones have been found to have a near limitless use in real-life applications. While drones have already been successfully applied to a variety of applications ranging from warfare, delivery and search and rescue, a number of emerging technologies are currently being applied in drone research to further accelerate the introduction of drones into the real world. Namely, artificial intelligence (AI) and nano- or micro-technology are very promising emerging technologies which are greatly increasing the realm of possibilities within the world of drones. Having the ability to make drones autonomous, intelligent and smaller has the potential to introduce drones into new environments to perform new real-life tasks, such as indoor exploration, indoor or outdoor non-destructive testing, and indoor target search.

A reason why larger drones have been applied in real life at a larger scale than smaller ones, is their higher computational and operational capacity. A higher computational or operational capacity allows for more brutal methods for controlling behaviors, such as simulataneous localization and mapping (SLAM) [1], wifi-positioning systems (WPS) or real-time locating systems (RTLS). Unfortunately, due to size and computational constraints, these methods cannot always be applied to smaller drones. This is especially the case if the environment has further constraints, such as unreliable or no GPS. Therefore, when working with small drones with limited computational capacity, alternative methods to control behaviors should be investigated.

All authors are with Faculty of Aerospace Engineering, Delft University of Technology, 2629 HS Delft, The Netherlands (e-mail: c.m.groen@student.tudelft.nl; s.li-6@tudelft.nl; g.c.h.e.decroon@tudelft.nl)

Innovations such as the Bitcraze Crazyflie [2] or the Delfly [3] were a very promising step towards working with miniaturized drones. Both are used by researchers and startups to research and develop new applications for drones. A variety of such applications have already been researched, particularly for swarming. A lot of this work is focused on state machines or other abstract modelling techniques. For example the SGBA [4], which is a minimal exploration algorithm designed for a swarm of Crazyflie's to efficiently explore a GPS-denied indoor environment, or the Learning to Seek concept [5] for source seeking using onboard reinforcement learning. Nevertheless, models such as a finite state machine (FSM) are considered to be abstract when representing logic, and are therefore difficult to understand or visualise. Furthermore, FSMs have a convoluted structure which makes them particularly hard to maintain during optimisation [6].

A useful alternative to FSMs are behavior trees (BTs). BTs have been largely applied in the gaming industry to define the behavior of non-playable characters. However, due to their simple and concrete structure with the ability to represent complex behaviors, are now being researched in robotics, including drones [7]. BTs are favored for their maintainability, reliability, and ease of understanding. This is particularly useful when attempting to optimise the behavior of a drone through non-manual techniques such as those applied in artificial intelligence [8].

Solutions to optimize drone controllers have been researched extensively for FSMs. However as discussed previously, optimizing an FSM can result in an optimized solution which is too abstract to understand [9]. This is particularly true for swarm applications, where the swarming in itself produces an abstract behavior. When considering BTs, a few optimization solutions also exist, but each of them have particular limitations. Numerical-based optimisation techniques would not suffice since they cannot optimize logic structures. A common option is to use genetic programming, which considers the full behavior tree in terms of subtrees that stretch out of different nodes. The structure can then be evolved to produce new trees by removing and adding subtrees onto other nodes in the behavior tree [6] [10]. This is however a computationally heavy option, which to a large extent also limits the novelty of the evolved BT. A novel technique known as grammatical evolution, inspired by genetic programming and biological evolution, offers a solution which allows for completely novel solutions, and much more flexibility in the search space of the optimisation

[11].

While grammatical evolution has been applied to behavior trees [8], these behavior trees are standardized for a specific task or behavior. This results in an inefficient framework which requires heavy modification of the parameters and structure of the grammatical evolution in order to apply them to various tasks or behaviors [8]. Therefore, a more generalized approach will be taken, such that the algorithm can evolve a variety of behaviors without any modification to the parameters or structure of the grammatical evolution. This generalized approach will focus on motion related tasks such as wall-following, navigation and exploration.

This new framework of evolving drone behaviors will be applied to wall-following, navigation and exploration, which are behaviors with a variety of applications including exploration and target detection, and varying complexities. Given a baseline controller, the grammatical evolution is expected to evolve a significantly better controller.

Following this introduction, section 2 discussed how behavior trees and grammatical evolution were applied, along with the chosen simulation environment, fitness and baseline BT. In section 3 the simulation results were presented, focusing on the wall-following evolution, and a navigation evolution. The setup and results of a real-life experiment, making use of the results from the wall-following evolution, were presented in section 4. Finally, recommendations into further work were given in section 5.

II. METHOD

This work mainly encompasses behavior trees and grammatical evolution. This section discusses how both of these were applied in simulation.

A. Behavior Tree

Behavior trees are becoming a viable model to represent and control drone behaviors, especially for developing and understanding novel behaviors.

1) Behavior Tree Structure: A behavior tree is a structure of nodes, starting from a root node (the parent node) which goes to at least one other node (the child(ren) of the root node). Apart from the root node, a behavior tree must have at least one leaf node, which has no children nodes, and only a parent node [9]. This leaf node must be a condition or an action. Beyond this, the behavior tree may have intermediate nodes, which have both parents and children nodes. The root node and intermediate nodes must always be in the form of a sequence or selector. A sequence node represents a node which returns an action only if all of its children nodes return 'True' for whatever conditions there may be. A selector node on the other hand returns the first child node which returns 'True'.

2) Baseline Wall-Following Behavior Tree: As an input to the grammatical evolution, a baseline wall-following behavior tree was created by hand. This hand-made controller was created and partially optimised through inspection, such that it is a reliable controller. The behavior tree can be found in Figure 1.

This behavior tree has a simple structure, where the root node is a selector node, and all of it's child nodes are sequence nodes. The children of every sequence node consists of leaf nodes. This means that this behavior tree will simply perform the action of the first sequence node which returns 'True'. Note that S0 represents the front sensor distance, S1 represents the left sensor distance, S2 represents the right sensor distance, and S3 represents the change in the left sensor distance over the course of 10 measurements. The first and fifth sequence nodes ensures wall-avoidance for the front sensor and dictates the wall-following direction, where the first one has no velocity for situations that the agent detects the front wall within 10 centimeters (S0 ¡ 0.1). The second and sixth sequence nodes ensure wall-avoidance for the left sensor (the side 'hugging' the wall), while the third and ninth sequence nodes ensure wall-avoidance for the right sensor. The fourth sequence node ensures detection of the end of a wall, such that the agent turns around the outer corner. The seventh node is responsible for keeping the agent aligned with the wall during the wall-following, while the eighth sequence node is meant for realignment with the wall in case the agent goes too far away from it. Finally, the last sequence node ensures that the agent continues moving forward when no other sequence nodes return true and there are no obstacles within 60 centimeters of the front sensor.

B. Grammatical Evolution

A subset of genetic programming, grammatical evolution takes inspiration from biological evolution in finding optimal solutions. Unlike other genetic programmes, grammatical evolution makes use of representing individuals through so-called genotypes and phenotypes. The resemblence to biological evolution is that the algorithm evolves the genotype (i.e. the 'DNA'), which is then translated into a phenotype (i.e. the behavior). In this simulation, the genotype was chosen to be a large collection of 8-bit strings, whereas the phenotype was represented as a behavior tree [12].

1) Grammatical Parser: As the name suggests, grammatical evolution makes use of a grammar to translate the genotype into a phenotype. In previous works, the grammar involved tailored conditions and actions for behavior trees, which resulted in grammatical evolutionary algorithms that are reserved for a small set of complex tasks [13] [11]. Instead, this paper uses a general grammar for the motion of agents. This grammar can therefore be used to optimise a large variety of problems related to motion (e.g. aggregation, wall-following, wall-avoidance, dispersion, exploration). The grammar used in this paper can be seen in Table I.

The logic behind this grammar involves terminal and non-terminal terms. Terminal terms are those that do not appear in the first column of the grammar, and make up the final behavior tree. Non-terminal terms, along with the genotype, are used to determine which terminal terms to choose. Within the grammar, every row represents a production rule, that tells the parser which translation to make based on the genotype, the rulesize (how many options does this production rule have), and the non-terminal term.

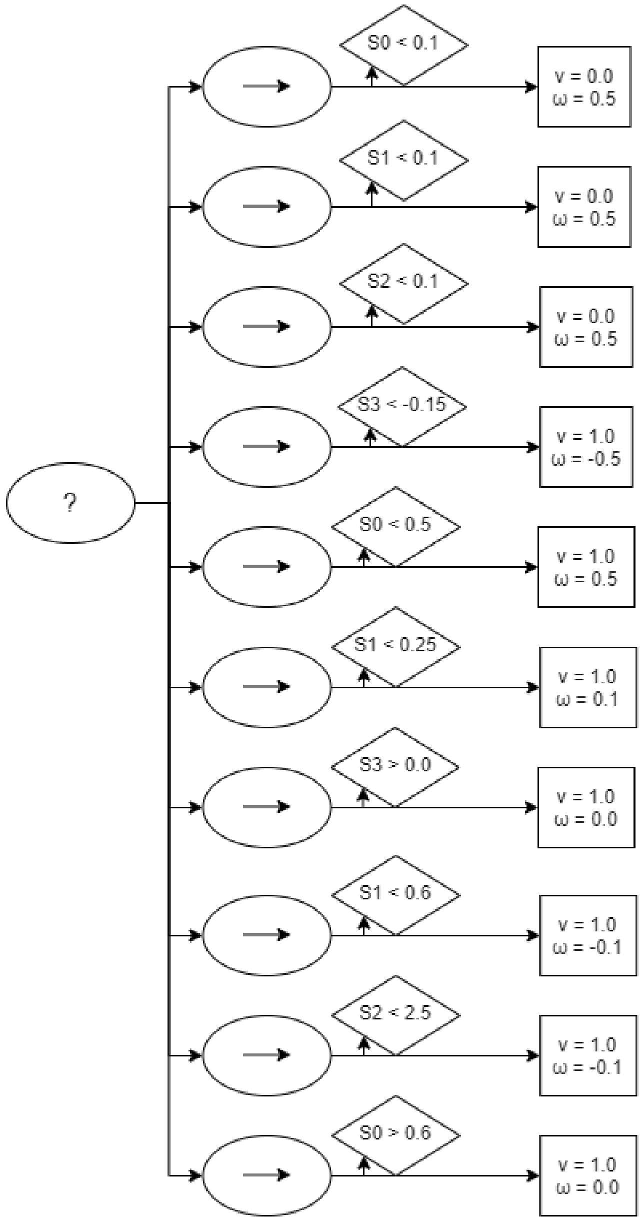

Fig. 1. The baseline wall-following controller. Note that the BT representation is rotated 90 degrees.

Figure 2 shows the logic of the parser, going from a string of bits to a behavior tree. Firstly, an M amount of 8-bit strings (the genotype) was translated into an M amount of integers. The representation of the behavior tree is then initialized using the non-terminal value 'ss'. The parser then loops through the M amount of integers, at each iteration taking the first non-terminal value in the representation of the behavior tree ('ss' at initialization), and selecting a new value from the grammar based on the modulus of the integer against the rulesize. The rulesize is based on how many choices there are for a particular production rule. For example, the production rule 'ss' has two choices ('sequences' or 'selectors'). In order to ensure that the behavior tree has the potential for growth, non-terminal terms such as 'sequence execution', 'ss ss' and 'conditions action' were included. With such choices, the

TABLE I
GRAMMAR USED FOR GRAMMATICAL EVOLUTION

Production Rule	Choice 1	Choice 2	Choice 3
ss	sequences	selectors	
sequences	sequence execution	ss ss	sequence
selectors	selector execution	ss ss	selector
execution	conditions	conditions action	
conditions	condition conditions	condition	
condition	greater	less	
action	speed&yaw		

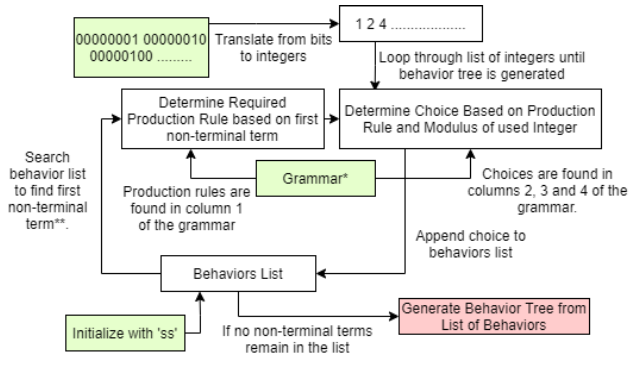
00000001 00000010 00000100 ....
Translate from bits to integers
1 2 4 ....
Loop through list of integers until behavior tree is generated
Determine Required Production Rule based on first non-terminal term
Determine Choice Based on Production Rule and Modulus of used Integer
Search behavior list to find first non-terminal term**.
Production rules are found in column 1 of the grammar
Grammar*
Choices are found in columns 2, 3 and 4 of the grammar.
Append choice to behaviors list
Behaviors List
Initialize with 'ss'
If no non-terminal terms remain in the list
Generate Behavior Tree from List of Behaviors

Fig. 2. Parser logic, going from a string of bits to a behavior tree. The green blocks show inputs into the parser, while the red block represents the output of the parser. *Refer to Table I for more information about the grammar. **A non-terminal term refers to a term which can be found in the first column of the grammar in Table I, and therefore corresponds to a production rule.

parser splits the choice into two terms and allows for growth in the tree. The parser may loop through the M amount of integers more than once until there are only terminal terms in the list of behaviors. For the purpose of this paper, 2000 bits are used.

One may note that the condition and action terminal terms do not actually specify what condition or action must be fulfilled. Therefore, whenever the condition or action term is chosen by the parser, the parser also uses the next two integers to determine the sensor type and threshold for the condition terminal terms, or the total speed and yaw rate for the action terminal term.

Furthermore, behavior trees have a structure which has to be defined within the parser. This is also done through the 2000 bit gene, but also through some BT logic. Firstly, whenever there is a sequence or selector term, the parser automatically knows that the next term has to go one step further from the root node. This is derived from the general logic of BT's. However if there is an action term, the third integer after this action term is used to determine how close to the root node the next term should come. Therefore, at every new integer the parser passes, the parser keeps track how far away from the root node the new term is, and if the new term is an action term, the third integer determines how much closer the following term should be to the root node.

Finally, in order to ensure a valid BT, the parser ensures that there is at least one action node in the generated BT. If there is no action node, the parser will place an action node at the end of the BT.

2) Evolutionary Mechanics & Parameters: Grammatical evolution drastically simplifies the evolutionary mechanics for a behavior tree, in that it only has to evolve the string of bits. This simulation makes use of the DEAP python library in order to evolve the string of bits.

The selection procedure was chosen to be tournament selection, with a tournament size of $0.06 \cdot population$. This means that for the selection, $0.06 \cdot population$ amount of individuals are randomly chosen from the population, and from these individuals the one with the highest fitness is chosen. This is repeated until the entire population is replenished. The population size was kept constant at 100 individuals in all generations [14]. This allows for a sufficient level of exploration, while at the same time limiting divergence in the evolution. In terms of mutation [15], an individual is randomly chosen (50% chance) and there is a 0.2% chance of a bit being flipped. Ultimately this means that on average 4 bits are flipped in every mutation. An individual in the new population also has a 25% random chance of being crossed over with another individual, and the crossover point is chosen randomly between 1 and 2000. In order to increase the pressure on the evolution of the population, a random map was generated for every new generation. Therefore, every individual was re-evaluated in every generation. More information about the random maps is given in subsection II-C.

In order to initialize the evolution, a baseline controller was made by hand. The string of bits representing this baseline controller was used to populate the initial population. This baseline controller is found in the previous section.

C. Simulation Environment, Fitness & Controller

The swarmulator environment [16] was used to evaluate the individuals within the evolution. Although the swarmulator was tailor made for swarms, this functionality was not used. Swarmulator was used due to the available wrapper to DEAP, and the ability to use a policy in the form of a behavior tree for the controller.

The controller itself uses the behavior tree to determine the inputs into the behavior of the agent. The controller also computes the data required for the conditions within the behavior tree, such as the sensor ranges and their derivatives.

The environments were custom made random maps with four different obstacles in a room of six by six meters. For every new generation, a new random map was generated. There were four obstacles which were placed randomly in each quadrant of the map. The five obstacles included a horizontal or vertical wall segment, a corner wall segment, a circular object, and a diagonal wall segment. During every evaluation of an individual, the individual is simulated for 300 seconds. During these 300 seconds, the individual must acquire the highest possible fitness. This fitness is based on the objective, which in this instance is wall-following. The fitness function is therefore dependent on the individuals distance to the walls, it's speed and a threshold distance to the wall which identifies a crash. The fitness function to determine the fitness F can be seen in Equation 1. Refer to subsubsection II-A.2 for an explanation of S0, S1 and S2. Also note that since this is a simulated environment, random noise was introduced into all of the sensor measurements, in order to account for the reality gap. Note that the units for S0, S1, and S2 are in meters. Also note that θ represents the angle turned in one direction to detect potential loops, in radian, and v_{tot} represents the speed of the drone in decimeters per second.

$$\begin{aligned} if \quad S1 < 0.6 \quad and \quad S1 > 0.2 \Rightarrow \\ F = F + 0.01 \cdot v_{tot} \cdot (0.6 - S1) \\ if \quad S0 < 0.1 \quad and \quad S0 > 0 \Rightarrow \\ F = F - 0.1 \cdot v_{tot} \cdot (0.2 - S0) \\ if \quad S1 < 0.1 \quad and \quad S1 > 0 \Rightarrow \\ F = F - 0.1 \cdot v_{tot} \cdot (0.2 - S1) \\ if \quad S2 < 0.1 \quad and \quad S2 > 0 \Rightarrow \\ F = F - 0.1 \cdot v_{tot} \cdot (0.2 - S2) if \quad \theta > 15 \Rightarrow \\ F = F - 0.01 \end{aligned} \tag{1}$$

III. SIMULATION RESULTS

In order to display the effectiveness of the grammatical evolution and behavior trees, a series of simulations were run. Firstly, the baseline wall-following behavior tree controller presented in Figure 1 was simulated. Afterwards, this controller was used in the grammatical evolution to evolve a new, better behavior tree. This new evolved behavior tree was simulated and visually inspected in order to identify particular differences in behavior to the baseline controller.

A. Baseline Wall-following Controller

The baseline wall-following controller was handmade and optimized through inspection of its performance over the course of a couple of runs. In a controlled and simple environment, this baseline controller had a consistent fitness score, and the behavior was logical and easy to understand. However, the performance of the baseline controller decreased significantly, and lost its consistency when exposed to more random and complex environments. The results for these simulations in more random and complex environments can be found in subsection III-C, combined with the results from the evolved controller, and another controller from the literature.

B. Evolution Performance for Wall-Following

The goal of this research was to create a grammatical evolutionary algorithm which could efficiently evolve a wall-following behavior. All evolutions were initialized with the baseline behavior tree presented in Figure 1, lasted for 100 generations, with each generation containing 100 individuals. A new environment was also initialised at every new generation.

Figure 3 shows a scatter plot of the evolution of the maximum fitness for four different evolutionary runs. A bestfit line was applied for every evolutionary run to better show the evolution of the maximum fitness. The four evolutionary runs are each represented by a unique color in the plot.

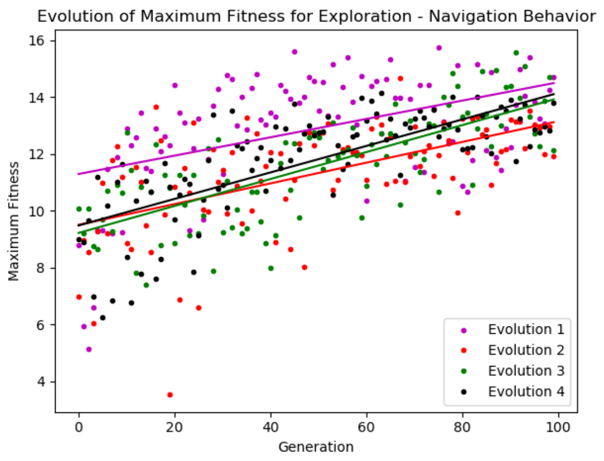

Fig. 3. Evolution of the maximum fitness for wall-following for four different evolutionary runs.

As one may see, each evolution has a positive increase in the maximum fitness, however some results are significantly different. Despite all of the evolution runs starting with the same baseline controller, evolution 1 has a significantly higher fitness than all other evolution runs in the first 50 generations. This is an indication that the evolution already evolved a significantly better controller in the beginning of the run. On the other hand, evolution 1 shows the least improvement in fitness. Due to the large search space of this problem, this issue could lie in the random approach to mutations and crossovers. According to Figure 3, evolution 1 experienced a fitness increase of 3.2 from the beginning to the end of the evolution, evolution 2 an increase of 3.6, evolution 3 an increase of 4.7, and evolution 4 an increase of 4.7.

C. Evolved Wall-Following Controller Performance

.

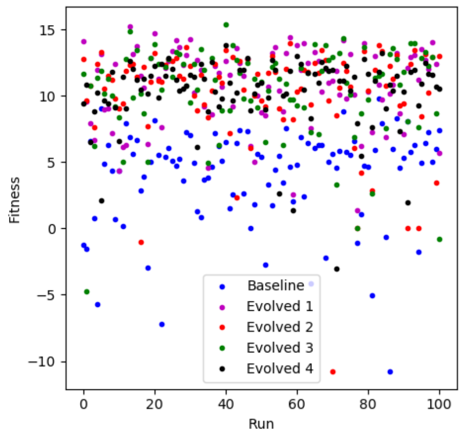

Fig. 4. Comparison of BT controller fitness performance for wall-following between the baseline and evolved controllers. Scatter plot representation.

Four separate evolutionary runs are shown in Figure 3. Every evolved behavior tree controller from each of these evolutions was compared to the baseline, as well as the other evolved controllers. This was done by running these five controllers (the baseline controller and four evolved controllers) on 100 unique environments, and thoroughly compare their fitnesses.

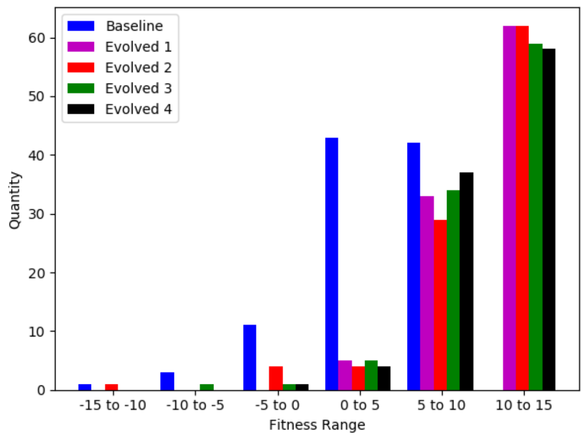

Fig. 5. Comparison of BT controller fitness performance for wall-following between the baseline and evolved controllers. Bar plot representation.

The raw results of this can be seen in Figure 4. One may directly notice the very wide range of fitnesses, from under -10 to over 15. This can be attributed to the level of randomness and complexity in the random room generator and the chosen fitness functions. Certain generated rooms consisted of 'traps', which were areas of the environment which were easy to get into, but significantly harder to get out of. Also, since the fitness functions involve a negative penalty for performing too many full rotations in a certain time frame, once an agent got stuck in such a 'trap', it would accrue a high negative fitness. This data was processed into a bar chart, seen in Figure 5 to better represent the distribution of fitnesses between the baseline and four evolved behavior trees. This bar chart allows one to notice that only evolved controllers scored a fitness higher than 10. Moreover, every evolved controller reached a fitness higher than 10 more than 50% of the time. On top of this, every behavior tree other than the first evolved behavior tree scored a fitness below 0 at least once. Note that the average fitness was 4.04 for the baseline, 10.81 for evolved 1, 10.18 for evolved 2, 10.43 for evolved 3 and 10.29 for evolved 4.

From this comparison, it could be deduced that the first evolved controller is superior to all other evolved controllers and the baseline. In the remaining analyses, this evolved controller will be considered. The evolved wall-following behavior tree controller can be seen in Figure 6. One can notice a few significant resemblances between the baseline and evolved controller, however there are also a few positive and potentially negative changes.

In terms of potentially negative changes, one can consider the unnecessary nodes which only result in bloating but do not contribute to the behavior of the drone. Firstly, the sixth

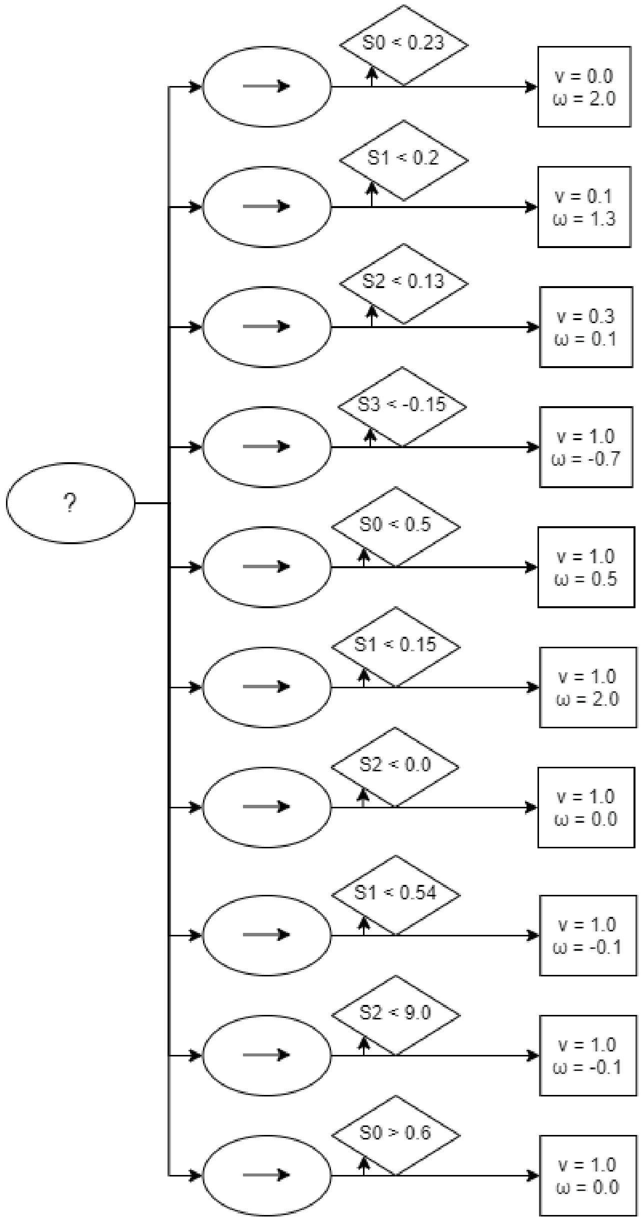

Fig. 6. The evolved wall-following controller. Note that the BT representation is rotated 90 degrees.

sequence node after the root node is never used. One may observe that this sequence node would not be called since the second sequence node would be called first. Furthermore, the seventh sequence node after the root node would also never be used, since it would indicate a range measurement less than zero, and therefore result in a crash.

Fortunately the evolved controller exhibits a few positive changes. The most substantial change is related to the seventh sequence node which is never used. In the baseline controller, this sequence node was responsible for keeping the agent aligned with the wall to the left at a specific orientation. Evidently this behavior in the baseline controller was unnecessary. Moreover, it can be deduced that this smooth alignment behavior greatly reduced the awareness of the agent. With only four available sensors to detect obstacles, it would be impossible for an agent moving forward to detect every possible obstacle. Removing this behavior resulted in a rapid scanning behavior while following the wall, which in turn allowed the agent to observe a wider view in front of itself. This scanning behavior was further complimented by significantly increasing yawing rates, by 20 times when aligning away from the wall, and 1.4 times when detecting a large increase in distance from the left sensor.

D. *Wall-Following Performance Comparison to Literature*

The chosen evolved wall-following behavior tree controller was simulated and visually observed in multiple environments against the behavior of a controller from literature and the baseline controller. The controller from literature was chosen to be the controller used within SGBA [4]. The environments for these three runs were chosen randomly.

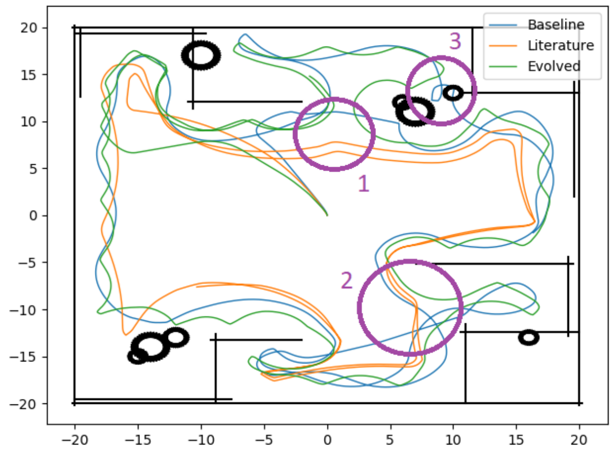

Fig. 7. First random environment simulation comparing the baseline, literature and evolved controllers.

Figure 7 displays the first analysed environment, where the points of interest are circled purple and numbered. In the first circle, one can note that both the baseline BT and literature controller failed to detect the turn left, most likely due to the presence of the circular object which caused them to divert. The same situation can be see in the second point of interest. In the third circle, one can see a clear collision, where the baseline BT got stuck in between the two circular objects.

Figure 8 shows the second environment with three points of interest as well. In the first circle, one can identify a collision between the circular obstacle and the baseline BT, where the evolved controller itself also almost collided. The third circle shows a very similar situation, where the evolved controller barely avoided the obstacle, however the baseline BT collided. Moreover this area also shows how the literature controller was not able to detect the area in the top right of the environment. Finally, the second circle once again shows a close call for the evolved controller, but also shows two collisions, one for the baseline BT, and the other for the literature controller.

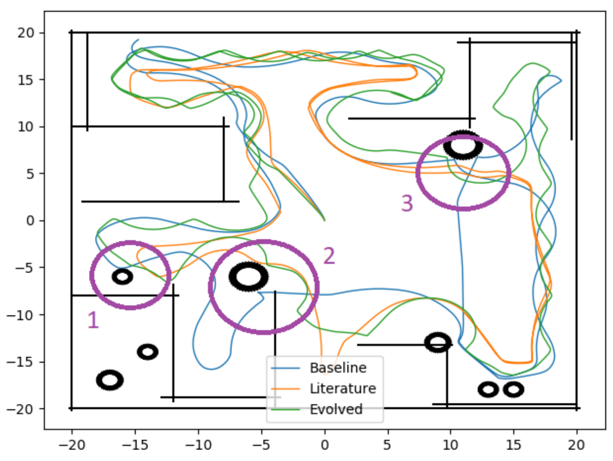

Fig. 8. Second random environment simulation comparing the baseline, literature and evolved controllers.

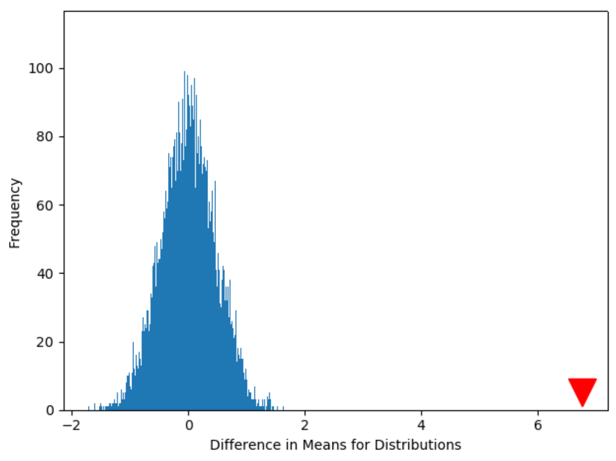

Fig. 10. Bootstrap analysis to determine significance of improvement.

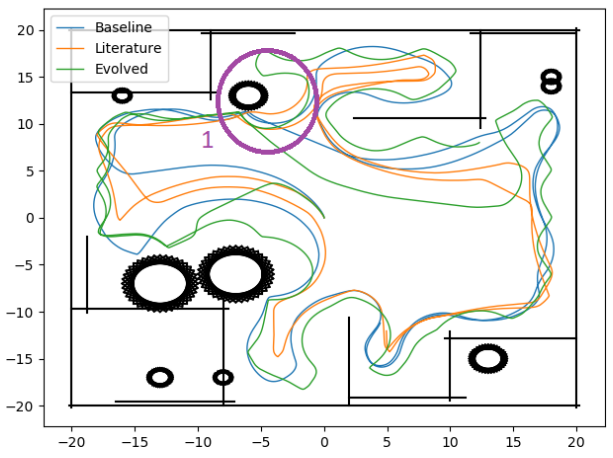

Fig. 9. Third random environment simulation comparing the baseline, literature and evolved controllers.

This final plot has only one point of interest. At this point you can see that both the literature controller and baseline BT collide with the obstacle. Although the evolved BT managed to just avoid the obstacle both times that it passed, it does come at the cost of it losing the wall in one of those instances, and therefore skipping a large section of the wall.

Figure 7, Figure 8 and Figure 9 all demonstrate (visually) the constant scanning behavior of the evolved BT. As mentioned before, it is this behavior that offers the evolved BT additional spatial awareness, and therefore a better chance of avoiding random obstacles.

E. Wall-Following Evolution Improvement Analysis

In order to get a more concrete understanding of the level of improvement in the wall-following performance, the results for the baseline and first evolved behavior trees from Figure 4 were analysed using the bootstrap method. From the comparison data, the mean fitness improvement between the baseline and first evolved behavior tree was 6.77.

The baseline and evolved behavior tree fitness data was then placed into a single data set, the combined data set. This data set was then resampled one hundred thousand times. During each resampling, two new data sets were created and both of their means were found. The mean from the second data set was then subtracted from the first data set. Since the combined data set was resampled one hundred thousand times, such a mean is calculated one hundred thousand times as well. These one hundred thousand variations of the mean were plotted in a distribution in Figure 10. The red arrow in this plot shows the mean fitness improvement from the non-bootstrapped results (6.77). As this plot shows, the mean fitness improvement of 6.77 is a significant improvement in the fitness.

F. Evolution for Exploration & Navigation Behavior

In order to complement the grammatical evolution results for evolving drone behaviors, a separate evolution was conducted for the exploration and navigation behavior. This behavior required the agent to explore a random environment and get as far away from it's starting position, and after a certain time limit (at half-time point) to navigate back to the starting position. For this evolution a few differences were introduced to the simulation, namely to the environment, fitness functions and the input sensors.

Due to the different task, a new environment generator had to be created. This new generator would randomly generate nine rooms of varying sizes and varying entrances to the different rooms. This allowed for the creation of an environment which is suitable for both exploration and navigation.

Furthermore, since the task is considerably different to wall-following, a set of new fitness functions had to be introduced. Ultimately, the new fitness functions would have to include a measure for exploration as well as navigation. For exploration, the fitness function involved measuring the furthest reached point in the environment for both the X and Y coordinates, shown in the first function in Equation 2. For navigation, the fitness function firstly measured whether or not the agent managed to find it's way back, and if it does, an additional fitness function accrued extra fitness until the end of the simulation to account for the efficiency of navigation. The fitness functions for navigation are seen in the second and third function in Equation 2.

$$
\begin{aligned}
&F = F + X_{max}/20 + Y_{max}/20\\
&if \quad starting - point \quad found \Rightarrow\\
&\qquad F = F + 10\\
&while \quad at \quad starting - point \Rightarrow\\
&\qquad F = F + 0.001\\
&if \quad S1 < 0.6 \quad and \quad S1 > 0.2 \Rightarrow\\
&F = F + 0.001 \cdot v_{tot} \cdot (0.6 - S1)\\
&if \quad S2 < 0.6 \quad and \quad S2 > 0.2 \Rightarrow\\
&F = F + 0.001 \cdot v_{tot} \cdot (0.6 - S1)\\
&if \quad S0 < 0.1 \quad and \quad S0 > 0 \Rightarrow\\
&F = F - 0.01 \cdot v_{tot} \cdot (0.2 - S0)\\
&if \quad S1 < 0.1 \quad and \quad S1 > 0 \Rightarrow\\
&F = F - 0.01 \cdot v_{tot} \cdot (0.2 - S1)\\
&if \quad S2 < 0.1 \quad and \quad S2 > 0 \Rightarrow\\
&F = F - 0.01 \cdot v_{tot} \cdot (0.2 - S2)
\end{aligned}
\tag{2}
$$

Finally, in order to promote a wall-following behavior throughout this task, the same functions as listed in Equation 1 were included, but at 10% of their weight in the wall-following task to ensure that the evolution was focused on evolving the exploration and navigation task. Also note that a fitness function for wall-following along the right sensor was included.

The values within this group of fitness functions were tuned to ensure a feasible distribution of fitness gain to evolve both exploration and navigation simultaneously. X_{max} and Y_{max} correspond to the furthest points reached during exploration for both the X and Y coordinate, in meters. Furthermore, the starting point was defined as a point within a 30 centimeter (0.3 units) distance of where the agent started.

In order to make the exploration and navigation tasks possible, an RSSI (received signal strength indicator) had to be implemented into the simulation. In order to bridge the reality gap, noise was included into the RSSI measurements. Due to the very fast responsiveness of the behavior tree and noisy RSSI, RSSI or it's derivatives were not used as a direct input sensor to the behavior tree. Instead, only an indication of the direction that the agent should go in based on the RSSI was given, represented by $RSSI_{Direction}$. This was done through finding the second derivative of the RSSI ($RSSI_{D_D}$). For exploration, since the agent would try to get as far away from the signal as possible, a negative $RSSI_{D_D}$ indicated a correct general direction to travel away from the source. Therefore, whenever the $RSSI_{D_D}$ turned positive, the $RSSI_{Direction}$ would change. For navigating back to the source, the opposite logic was used (negative $RSSI_{D_D}$ resulted in a change in $RSSI_{Direction}$).

Furthermore, as mentioned before, the exploration and navigation task would require the agent to be capable of following the wall along it's left and right side. Therefore an additional sensor, $Wall_{Direction}$, was implemented. This sensor would simply dictate whether the agent should follow the wall along the left or right, by changing the yaw direction when approaching a wall from the front. This implementation was required to avoid the drone from getting stuck in a loop, if it were in a room where the direction to the exit did not correspond to the preferred RSSI direction or the wall-following orientation. By simply changing the wall-following direction, the agent would be able to get out of the room. Therefore, some very rough yaw-based odometry was implemented, such that if the agent would make two full rotations or more in one direction, the wall-following direction would change. This however only solved the issue in most cases, since in some rooms, both wall-following directions could result in looping. Therefore, whenever the wall-following direction changed, the agent would be forced to follow the wall for forty simulation seconds before being able to follow the RSSI Direction again. This does unfortunately reduce the flexibility for optimising the behavior through grammatical evolution.

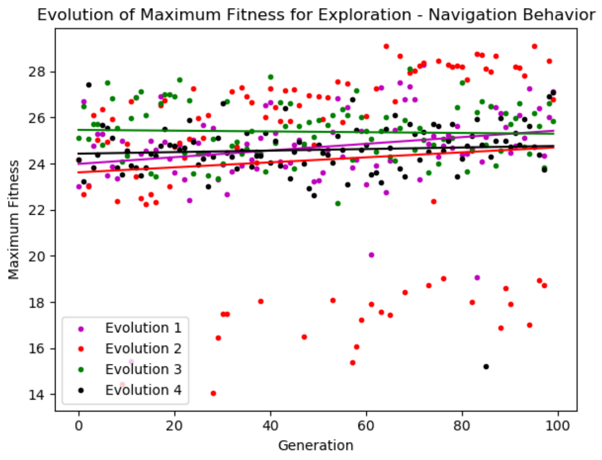

Fig. 11. Evolution of the maximum fitness for the exploration and navigation behavior.

The results for the evolution of this task are shown in Figure 11. Due to this considerably more difficult task, it is evident that the slope of the best fit line for all four evolutions are much lower. Notably, the third evolution even displayed a negative slope, which would indicate devolution.

The evolved exploration and navigation BTs which were generated in Figure 11 were compared against the baseline BT on 100 randomly generated environments. These results can be found in Figure 12. The results were relatively poor, where only the second evolved BT scored on average higher than the baseline controller. The average fitness for the baseline was 21.15, for the first evolved BT 18.98, for the second 23.79, for the third 12.92 and 18.62 for the fourth evolved BT. The result for the third controller was expected, given the poor evolution progress as shown in Figure 11

Figure 13 displays the fitness distribution for the five different BTs. A few observations to note. Firstly, this

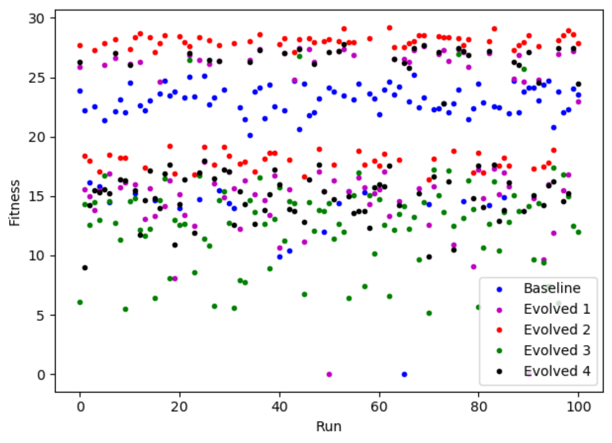

Fig. 12. Comparison of BT controller fitness performance for exploration and navigation between the baseline and evolved controllers. Scatter plot representation.

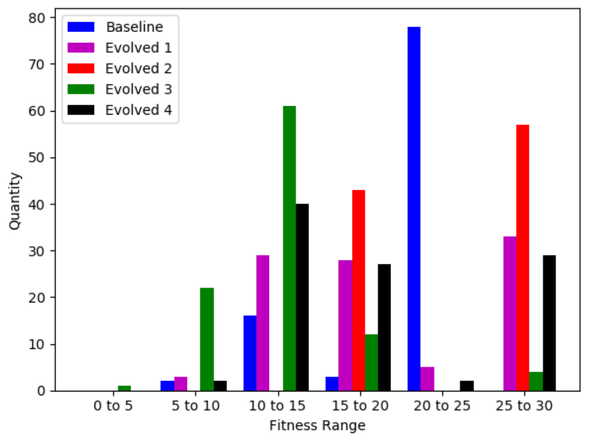

Fig. 13. Comparison of BT controller fitness performance for exploration and navigation between the baseline and evolved controllers. Bar plot representation

distribution plot allows one to notice the poor performance of the first, third and fourth evolved BTs. These three evolved BTs all had a significant number of runs where the agent only scored a fitness of 10-15, and a few even in the 5-10 range. Interestingly, the second BTs score was almost evenly divided between a range of 15-20 and 25-30, with no run scoring a fitness below 15 or between 20 and 25. This does indicate a significant flaw in the second evolved controller. After analysing the second evolved BT, it was determined that the evolution focused on evolving an efficient wall-follower in this case, rather than an efficient navigator. This controller would therefore only follow the wall, and find the source location only 60% of the time (includes all scores between 25-30), compared to almost 80% for the baseline. Therefore, despite receiving a higher average score over the 100 random runs, the second evolved controller is unfortunately ineffective with navigation. Recommendations for improvement are listed in section V.

IV. EXPERIMENTS

Following the simulations, real-life experiments were performed to evaluate the performance of the evolved controller in practice. It is also useful to detect potential reality gaps, and to compare it to the existing wall following controller from literature. Unfortunately due to limited resources, which included a lack of time and insufficient hardware to reduce the RSSI noise or to use another system, only the wall following behavior tree was tested in real life.

A. *Experimental Setup*

In order to perform the experiment, a Crazyflie drone was used. This Crazyflie was equipped with a a multiranger deck, which features four IR sensors (front, left, right and back) for range detection. It also made use of the Flow Deck V2 for optical flow measurements.

Due to this drone of choice, the environment had to be very specific and controlled. The Flow Deck V2 can only be used on bright and patterned floors. Furthermore, the available flying time had to be reduced to approximately three minutes, to ensure sufficient power in the drone.

The literature controller used for this drone was the same one that was used for the simulations. The software for this drone can be found within the GitHub repository for the Crazyflie firmware [17]. The evolved controller used was almost identical to the controller evolved in simulation. Note that a few changes had to be made due to the reality gap, specifically due to the drone dynamics. These are described below.

It was observed that the movement of the Crazyflie can be erratic and that drift was common, especially in imperfect conditions due to limitations of the Flow Deck. Furthermore, at higher speeds the dynamics of the Crazyflie resulted in suboptimal reactions to sensor inputs (e.g. despite setting a sensor distance of 30cm, the Crazyflie would only react to this input at 15-20cm). Unfortunately this was not accounted for in the simulations. Therefore, it was decided to increase the lower sensor distances to allow for more clearance from the wall and obstacles. Finally, this insufficient reaction time in the dynamics also negatively impacted the Crazyflie's ability to detect outer corners through the difference in the left sensor distance. The solution to this was to double the yawing rate in these situations, which helped keep the Crazyflie aligned to the outer corner of the wall.

B. *Validation Results*

Figure 14 shows the path travelled for the evolved wall-following BT, while Figure 15 shows the path travelled for the literature controller. The environments for both cases were identical, featuring four walls and four obstacles (plastic house plants).

One may identify a notable difference between the results, in that the literature controller has a much smoother path. While this does produce a more elegant path, it proved to greatly diminish the spatial awareness of the agent. In the literature controller, by moving forward without constantly scanning what is ahead, the agent struggled with the four

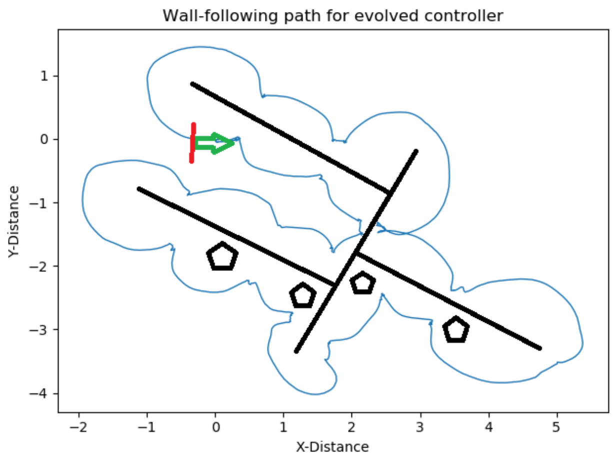

Fig. 14. Real-life experiment path taken by the evolved BT.

obstacles that were placed close to the walls. It took a total of four runs before the literature controller was able to avoid an obstacle which was placed around the X = 1.5 and Y = -2.0 point, and another one around the X = 3.8 and Y = -2.0 point. For both obstacles the agent would either not detect it at all, and fly into it with one propeller, or detect it far too late and yaw into it with one of the propellers.

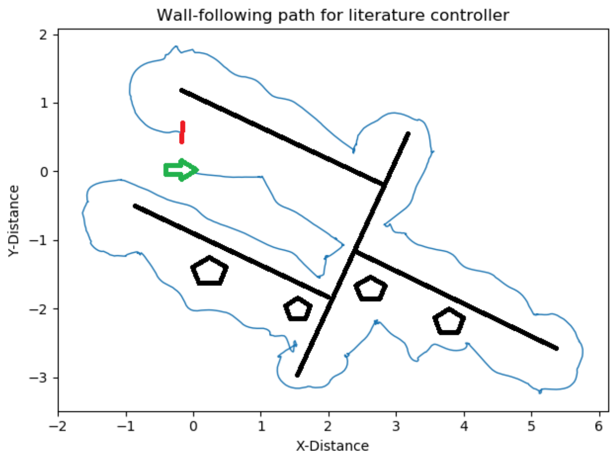

Fig. 15. Real-life experiment path taken by the controller from literature.

Moreover, despite the evolved controller requiring to stop in some instances, the overall motion is a lot more continuous when compared to the literature controller. This resulted in a far more efficient wall-following behavior, where the evolved BT managed to return to its starting point in one minute and thirty nine seconds, the literature controller took over two minutes and two seconds.

V. RECOMMENDATIONS

The focus of this paper was to determine the eligibility of using grammatical evolution to evolve BTs for drone behavior. This was done in the scope of keeping the BT as low level as possible, such that the actions corresponded to speed and yaw outputs, rather than higher level behaviors. This allowed for the simulataneous evolution of lower level parameters (conditions and actions), along with the overall behavior (structure).

A. *Large Search Space*

As one may observe in subsection III-C, the grammatical evolution unfortunately focused on the evolution of the conditions and actions, whereas there was little change to the structure where only one sequence node became obsolete. This can be explained through the incredibly large search space, where an individual could be represented through any combination of 2000 bits. This resulted in a search space of 2^{2000} (a value with over 600 digits), whereas one evolution searched through a maximum of 10,000 solutions. Realistically, due to duplicate individuals, the maximum number of searched solutions per evolution was likely to be much lower. Such an evolution already took eight hours with the available resources.

One may assume that running evolutions with higher populations and more generations on a better equipped computer, could in fact generate much more novel and interesting results. Another option would be to reduce the number of bits corresponding to each integer within the grammatical evolution. Currently, one integer is represented through eight bits, which can give it a value between 0 and 255. As mentioned, this was required to give enough flexibility for the optimisation (through evolution) of the low level parameters such as speed, yaw and sensor distances. If one integer could be represented through three bits, the search space would already be over 10^{375} times lower. This would however mean that the sensor distances, speed, and yaw could only have 8 different values, rather than 256.

B. *Population Management and Tracking*

Another significant limitation of this work can be found within the evolution procedure itself. There is a level of inefficiency in the way the populations are managed between generations, which is due to a lack of information sharing between the different generations and the overall complexity of the problem.

Firstly, the best BT chosen at the end of the evolution is solely based on the last generation. Therefore, there is a chance that the chosen BT is only the best for that particular environment, but is suboptimal in most other environments. This issue could be minimised by testing every individual on multiple environments within one generation, or by choosing the best BT based on the performance of BTs in previous generations as well.

Furthermore, this lack of information sharing between generations meant that poor BT structures could be repeated regularly after random mutation or crossover. Although 2,000 bits were used to represent an individual, in many cases less than 1,000 bits were required to build a BT. Very poor BT structures are those that do not have a single action or condition node, or are heavily bloated through selector or sequence nodes - where many of these types of BTs can be identified through the first 100 bits. Although this would make the evolution more computationally heavy, the algorithm could reduce the randomness of mutations and crossovers by avoiding crossovers or mutations that lead to a poor BT structure.

References

[1] L. Jaulin, “Range-only slam with occupancy maps: A set-membership approach,” *IEEE Transactions on Robotics*, vol. 27, no. 5, pp. 1004–1010, 2011.

[2] W. G. et al, “Crazyflie 2.0 quadrotor as a platform for research and education in robotics and control engineering,” 2017.

[3] C. de Wagter and J. Mulder, “Vision-only control of a flapping mav on mars.” American Institute of Aeronautics and Astronautics Inc. (AIAA), 2007. [Online]. Available: https://research.tudelft.nl/en/publications/vision-only-control-of-a-flapping-mav-on-mars

[4] K. M. et al, “Minimal navigation solution for a swarm of tiny flying robots to explore an unknown environment,” *Science Robotics*, 2019.

[5] B. P. Duisterhof, S. Krishnan, J. J. Cruz, C. R. Banbury, W. Fu, A. Faust, G. C. H. E. de Croon, and V. J. Reddi, “Learning to seek: Autonomous source seeking with deep reinforcement learning onboard a nano drone microcontroller.” *CoRR*, vol. abs/1909.11236, 2019. [Online]. Available: http://arxiv.org/abs/1909.11236

[6] K. S. et al, “Behavior trees for evolutionary robotics,” *Artificial Life*, 2016.

[7] M. I. et al, “A survey of behavior trees in robotics and ai,” *IEEE International Conference on Futuristic Trends in Comupational Analysis and Knowledge Management*, 2020.

[8] A. N. . M. Goodrich, “Designing emergent swarm behaviors using behavior trees and grammatical evolution,” *AAMAS 2019*, 2019.

[9] M. O. et al, “Behavior trees for decision-making in autonomous driving,” 2016.

[10] A. F. W. et al, “Onboard evolution of understandable swarm behaviors,” *Advanced Intelligent Systems*, 2019.

[11] E. F. et al, “Geswarm: Grammatical evolution for the autimatic synthesis of collective behaviors in swarm robotics,” *GECCO*, 2013.

[12] M. O. et al, “Grammatical evolution,” *IEEE Transactions on Evolutionary Computation*, vol. 5, no. 4, 2001.

[13] A. N. et al, “Geese: Grammatical evolution algorithm for evolution of swarm behaviors,” *GECCO*, 2018.

[14] A. S. et al, “Comparative review of selection techniques in genetic algorithm,” *IEEE International Conference on Futuristic Trends in Comupational Analysis and Knowledge Management*, 2015.

[15] F. S. et al, “Evolutionary robotics,” *Scholarpedia*, 2016.

[16] M. Coppola, “Swarmulator.” [Online]. Available: https://github.com/coppolam/swarmulator

[17] K. McGuire, “Crazyflie wall following demo.” [Online]. Available: https://github.com/bitcraze/crazyflie-firmware/tree/master/examples/demos/app_wall_following_demo