

Push Implementation of Biped Locomotion Control

Deliverable 1: Final Year Dissertation

Jacob Cooper - H00251723 | Dr Michael Lones | Computer Science|

Contents

[1 Introduction 2](#_Toc24569117)

[1.1 Aim 2](#_Toc24569118)

[1.2 Objectives 2](#_Toc24569119)

[1.3 Stakeholders 2](#_Toc24569120)

[2 Background 3](#_Toc24569121)

[2.1 Evolutionary Algorithms 3](#_Toc24569122)

[2.1.1 Genetic Algorithms 4](#_Toc24569123)

[2.1.2 Genetic Programming 5](#_Toc24569124)

[2.2 Push 13](#_Toc24569125)

[2.2.1 PushGP 16](#_Toc24569126)

[2.2.2 Push3 16](#_Toc24569127)

[2.2.3 Linear Push (Plush) 17](#_Toc24569128)

[2.2.4 PyshGP 18](#_Toc24569129)

[2.3 Evolutionary Robotics 20](#_Toc24569130)

[2.3.1 Applications in Genetic programming 21](#_Toc24569131)

[2.4 Webots 21](#_Toc24569132)

[2.4.1 Application of Webots 22](#_Toc24569133)

[3 Requirements 23](#_Toc24569134)

[4 Design 24](#_Toc24569135)

[5 Evaluation Strategy 25](#_Toc24569136)

[6 Project Management Plan 2f5](#_Toc24569137)

[6.1 Timetable 25](#_Toc24569138)

[6.2 Risk Analysis 26](#_Toc24569139)

[7 References 27](#_Toc24569140)

# 1 Introduction

## 1.1 Aim

Investigate the use of Pushgp for creating robot controllers

## 1.2 Objectives

* Explore morphology of robot
* Create a suitable testing environment
* Use pushgp to explore controllers
* Invvestigate the various implementations of Push

## 1.3 Stakeholders

* Supervisor (Dr Michael Lones)
* Heriot-Watt University
* Robotics Researchers
* Game Developers
* Companies which use robots
* Creators of any code libraries I use
* Creators of Webots
* Users of Webots
* Genetic Programming researchers
* Push researchers

# 2 Background

## 2.1 Evolutionary Algorithms

Evolutionary Algorithms (EA) take influence from nature and biology to try and solve problems.[2] It does this by employing strategies used in nature, such as evolution. Examples of this would be Genetic Programming and Genetic Algorithms. Most EA’s are either :

* Generational: meaning they create an entire new set of individuals each iteration [2]
* Steady-state algorithms: meaning they choose a subset of individuals to iterate over at a time [2]

Although Genetic Programming (GP) and Genetic Algorithms (GA) are slightly different, they do (generally) share a few fundamentals on how we form them. Both generally: create a set of candidate solutions which we can test, the best solutions are then used to derive further solutions (maybe generational or Steady-state). We then repeat this cycle until either we find a suitable solution, or we run out of time. The methods described above have names and are a part of almost all GA’s and GP’s they are [1] [2][7][17]:

1. Initialisation of the population
2. Fitness Assessment to find the fitness of each individual
3. Fitness Selection to select the fittest individual in a population
4. Crossover and mutation of the fittest individuals

We then repeat steps 1 and 2 for the next iteration until we find a solution found or we run out of time. There are many variations on how to initialise the population, how to assess their fitness, and how to do crossover and mutation [1][2][7][17].

Some Algorithms may replace steps with another of their own or miss out on some steps completely. We will touch on Genetic Algorithms, but our focus here is on Genetic Programming. Below are attributes that exist for all EA’s. Further down (Sections 2.1.1 and 2.1.2) we will discuss the procceses and properties that are unique to certain EA’s and their representations.

#### Initialisation

When starting a population, the most significant decision is how we represent our population. A few different representations exist depending on which EA we use and different initialization processes exist for each of these representations [1][2][7][17].

#### Fitness Evaluation and Selection

Our choice in fitness evaluation and selection can have substantial consequences on what solutions we put out [1][2]. These processes are done as follows:

1. Fitness Evaluation: Each individual’s fitness is assessed usually based on a fitness function. This function considers multiple variables to sense how “well” a program has done at its given task.
2. Fitness Selection: Process in which we pick individuals Crossover and Mutation. Usually, fitter individuals are more likely to be picked.

Fitness Evaluation and Selection is universal across all EAs, meaning the methods we describe next can be used for any EA.

##### Tournament Selection

Tournament selection is one of the most popular methods for fitness selection in many GA’s and even GP’s.[1][2][17] There are several reasons for this, such as:

* it is not very sensitive to how the actual fitness function works,
* it is straightforward,
* requires no preprocessing
* works well with parallel algorithms (necessary for running as many iterations as possible at once),
* Is very modifiable (able to change the size of the tournament as well as how selective it is).

Tournament selection works by first selecting a random group of individuals in a population. Then iteratively comparing them everyone in this group with another in a tournament fashion until we have an ultimate winner in that section of the population. We do this across the whole population by splitting it into n groups, having n tournaments and then using the winners of those tournaments for crossover and mutation.

##### Lexicase Selection

First introduced by Lee Spector (2012)[3], lexicase selection considers test cases separately rather than an aggregate like Tournament Selection. It begins with a pool of possible parents. These parents are slowly narrowed down through many different fitness cases which are applied one at a time. We can find many different methods of lexicase selection by altering the criteria to be in this initial pool.

The most basic form of Lexicase Selection is called “global pool, uniform random sequence, elitist lexicase selection”. It has an initial pool that is the entire population; this is called “global pool”. We apply The fitness cases to each parent in a random order. We select from a uniform distribution of all possible orderings; this is called “uniform random sequence”. Only the individuals with the best fitness for a given fitness case can survive onto the next fitness case; this is called “elitist”.

##### Epsilon-Lexicase Selection

Introduced in 2016 (Lee Spector et al.) [4], Epsilon-Lexicase Selection works in much the same way as Lexicase selection in that it has a pool of parents and a set of fitness cases. The difference is that the pass conditions for each test case are more effectively defined.

Epsilon-Lexicase in four steps:

1. Initialise the pool as being the entire population.
2. Shuffle the fitness cases
3. Use elitism to select only individuals with the best fitness in each fitness case to proceed to the next fitness case.
4. If there is only one parent left, then this is our selected parent. If there is more than one parent left in the pool, then we repeat step 3 with the next fitness case. If there are no more fitness cases left and still more than one parent, then a parent is randomly selected from those remaining in the pool.

#### Crossover

Crossover is considered a macro genetic operation [7] . The purpose of crossover is to find two individuals and swap some parts in them. We do this because there is usually a good trait in both parents that allowed them to be selected. We hope to combine them to get a good trait from both parents to create an even fitter child [1][2][7][17].

#### Mutation

Mutation is considered a micro genetic operation [7]. Mutation is done to keep variation in the population. It is essential to keep our population diverse in order to stop our algorithm prematurely coming to a halt. It may think it has found the optimal solution. However, this solution may be a local optima and not the most optimal solution for the problem (global optima). [1][2][7][17]

### 2.1.1 Genetic Algorithms

John Holland conceived the first idea of genetic algorithms at the University of Michigan in 1970. This GA selects individuals one group at a time. Then it generates children from each of these groups until enough children have been created to fill the next generation. Typically this is performed over fixed-length Boolean vectors. A GA can be represented using vectors of Boolean values, real values or integers. Below we will discuss the various processes and properties that are unique to GA’s

#### Crossover

Crossover in GA’s swaps value in two vectors with eachother. There are a variety of ways to do such this, one such way is One-Point Crossover.

##### One-Point

Picks a number between one and the length of the vector (inclusive), then swaps any indexes less than this value. There is a chance that this number is equal to one, this happened at a probability of 1/length, and when it happens, there is no crossover.

The problem with this method is that the first two vectors (v1 and v2) are highly unlikely to be split, whereas the first and last are highly likely to be broken up. This crossover struggles to find a solution requiring a secure connection between the first and last vector. Alternatively, a solution that requires that we split the first two elements often will also be challenging to find.

It is important to note that crossover cannot get every conceivable solution. Crossover only produces children which preside in a ‘box’ which cannot ever increase. To counter being confined to a box, we perform mutation.

#### Mutation

Mutation in GA’s are done by selecting an element in the vector (or multiple) and changing their value. If the GA is represented by a binary vector then mutation can be simple done with Bit-Flip Mutation.

##### Bit-Flip Mutation

Start by going down the vector and for each Boolean in the vector we give it a chance of (normally 1/vector length) to flip its value. Bit-Flip Mutation is a very simple mutation that can work quite well, but the drawback is that it only works on binary vectors.

### 2.1.2 Genetic Programming

Genetic programming focuses on finding optimal small computer programming which exist in a sizeable space of possible computer programs. GPs work when there is a small space of possible programs, but it is not clear which ones are optimal. Representations are usually lists or trees. These lists/trees are all usually formed from a set of essential functions or CPU operations. The operations involved in initialisation and modification in GP’s are mostly for maintaining closure. Closure is the production of valid individuals from previous valid individuals.

#### Linear GP

We can represent GPs as arbitrary-length lists of machine language instructions. Closure must be maintained while executing arbitrary machine code strings. It is essential to construct the list from a carefully selected set in order to maintain closure. For Example: If instructions are finite in length, then we can assign a unique integer to each instruction. There is usually uses a finite set of registers. These registers cause the machine code lists to operate more like directed acyclic graphs. Early instructs will affect instructs much later due to them sharing a register. Registers are selected from a register set and can either be Input registers or calculation registers. Input registers hold inputs for the program before execution. Calculation registers typically initialise with a constant and may be used to store the results of some calculation. Output registers can be defined from input registers or even calculation to store the output from a program. Another vital part of Linear GP is the instruction set. The instruction set is essentially the programming language used in the evolution of the programs. Linear GP is usually used to represent imperative or machine languages.

##### Initialisation

Generation of lists depends on what domain we require them for. However, there are two general issues when initialising a list: the length of the list and populating it. Usually, Initialization is entirely random. An upper bound must be defined, and a lower bound must be defined. The lower bound may be equal to one instruction. Then when we create a new program with length uniformly chosen from between the previously defined bounds. It is essential not to make the initial length of programs too large or too small. Too large of a program will reduce the variability that can happen during evolution. Too small of a program can lead to programs not being diverse enough and potentially converging too early.

##### Crossover

Crossover is done in Linear GP taking one or more values from either parent and swapping them. We can do Crossover via the “two-point linear crossover” method.

###### Two-Point Linear Crossover

We start by taking a parent and selecting two points between zero and the length of the parent. Then we take the other parent and doing the same then swapping the values inside these two ranges. One problem is that a child may become too large if the sections selected are not equal. One way to counter this would be to ensure when selecting sections of the parents, that the sections are of equal length.

##### Mutation

Although mutation is not as common in GP as it is in other EA’s it is still possible (and even recommended) to prevent premature conversion.

We can do Linear GP Mutation in many ways, such as:

* Randomly replacing an instruction with another, that takes in the same number of inputs
* Randomly replacing a constant with another of the same type
* Randomly replacing a register with another register

If we use only one genetic operator, then it is likely that the variation will not be high. The evolution will progress more precisely with smaller changes throughout each iteration and would be more exploitative. However, if we use multiple genetic operators at once, then it will make much more significant changes between generations and is more explorative. Usually, micro mutations and macro mutations are both used each iteration; an example of this would be Crossover and Mutation as explained above.

#### Tree- Style representation

(add in extra here) We may restrict some of the nodes of tree representations to how many children they can have; this is an essential quality for the construction of trees in initialisation.

##### Initialisation

To produce a tree: the GP selects iteratively from a function set. In this function set, each member has a set arity. Nodes with zero arity are leaf nodes, and nodes with greater than or equal to one arity are either a branch node or the root node. In order to build a valid tree, we must follow these rules.

One conventional algorithm for initialisation is the Grow algorithm. In the “Grow” algorithm, we build a tree is from the function set in a depth-first manner up to a maximum depth. Another algorithm “Full” is very similar to “Grow” except the only difference is “Full” forces the tree to be maximum depth whereas “Grow” does not.

An algorithm that strikes a balance between “Full” and “Grow” is “Ramped half-and-half”. “Ramped half-and-half” gives a 50% chance to “Full” and “Grow” and usually randomly selects depth from 2 to 6.

###### Ephemeral Random Constants

Sometimes we need several random constants added to the tree. Function sets do not have to be of a fixed size; sometimes, we can include a particular node called the Ephemeral Random Constant (ERC). When we pick an ERC, it transforms into a randomly generated constant of our choosing. Once this has happened the value of the ERC can not be changed except through unique mutator methods.

##### Crossover

In Tree-style GP,Crossover is done by swapping subtrees between two parents.

The three most basic ways of doing this crossover are:

* Standard Crossover
* One-Point Crossover
* Uniform Crossover

###### One-Point Crossover

“One-Point Crossover” as proposed by Poli and Langdon (1997) works similarly to Standard crossover in that it picks one point and swaps the corresponding subtrees. It is typically done in two stages to account for diversity between trees.

1. Traverse both parent trees to find nodes of same arity along a path which also has the same arity (mainly looking for parts in the parents which have the same shape)
2. Select one random link with uniform probability along this path and swap the subtree after it.

One of the essential properties of the One-Point Crossover method is “it makes the calculations necessary to model the distribution of GP schemata feasible”. Essentially, it is easier to see which area of space is likely to be searched by the next generation.

It is important to note that without mutation One-Point Crossover converges quite quickly and so to keep this from happening we must mutate.

##### Mutation

Mutation is done is Tree-style GP by selecting one or more nodes to be randomly changed. The two most basic forms of mutation in Tree representation GPs are Subtree mutation and point mutation.

###### Sub-Tree Mutation

Starts by picking a random mutation point in a given tree. Then swap this node and its subtree with a randomly generated subtree using an algorithm such as Full or Grow. Normally, Grow with a max depth of 5 is used for the generation of the subtree. Sub-Tree Mutation suffers from a similar problem as standard crossover as mostly leaf nodes are picked and so not much is changed. To remedy this, we do the same as before: we pick branch nodes 90% of the time and leaf node 10% of the time.

###### Point Mutation

We chose a random mutation point in the Tree. Take the value of this node. If it is a branch node, replace it with a value from the function set that has the same arity. If it is a leaf node, replace it with a value from the terminal set. For example, a “+” operation could be replaced with a “\*” operation as they both have an arity of two or a “1” could be replaced with a “2” as they both have zero arity (e.g. are leaf nodes). If no function matches the arity of a selected node, then nothing happens. Subtree mutation affects one tree, whereas point mutation is on a per-node basis.

## 2.2 Push

Push is designed to have a completely uniform syntax so the ease the process of generating code that manipulates code (say for auto constructive evolution). Push also supports a multitude of different data types as well as recursion and subroutines. It is an extension of stack-based programming languages such as Forth(Salman 1984).

A push program consists of a string of instructions, constants and parenthesis. There is only one syntax rule: we must balance the brackets. The instructions found in these strings take their arguments from a Global stack. Each data type has one global stack and instructions push their outputs onto these global stacks. It is written in postfix notation, meaning that for a human might be a little challenging to read it (more on this later). When implementing the evolution of multiple data types, we would typically use strongly typed genetic programming. The problem with this is there is substantial syntax restriction imposed on passed data to ensure the types match what they are supposed to be. This problem can limit options for crossover and make the evolutionary process far more complicated than It needs to be. Push breaks free of these problems with its uses of stacks.

Push includes a stack called the “exec” stack which programs push to for execution. This stack contains all the program code still to be executed at any point during execution.

We can skip some instruction (such as ‘+’) if there are not enough arguments on the argument stack. So for example if a program reads ‘integer’ it will be stored in the type data stack, then if a ‘+’ method appears it first looks at the type stack to see what type ‘+’ is. The system does this because ‘+’ could have multiple types (e.g. integer/string). Is there is no type on the stack then it will resort to a default. This default has a list of types consisting of Integer, Boolean, code, type, name in that order from top to bottom. From this default, it will pick the highest type applicable. So an appropriate type will always be found for a push operation.

There are a few operations implemented for all data types: pop (return the item on top of the stack), dup (duplicate item on top of stack and push it to stack) and swap (swaps the two highest items in the stack).

The parenthesis in a push program are only there for structure and do not have any real effect on execution.

Push also allows for code manipulation through its ‘code’ data type, which inherits from the ‘expression’ data type.

A vital instruction in code manipulation expressions is the ‘quote’ instruction. When we execute the ‘quote’ instruction, We will push the next piece of code to go through execution will to the code stack rather than being executed.

The “do” instruction recursively calls the expression on top of the “code” stack, mostly it is a way of executing the code on the “code” stack which was put there by the “quote” instruction. A variant of “do” is “do\*” which does the same operation except it pop the code from the “code” stack first.

Code is pushed to the “code” stack before execution when first submitted to the interpreter. We can then use the “quote” and “do” instructions to implement recursion.

The “if” instruction executes the top two items on the code stack. It also checks the top element of the “Boolean” stack. Depending on what value it finds there, it may discard the rest of the code stack.

Due to all the instructions which cause recursion, we need guards against non-termination. To accomplish this, we put a limit on how many instructions can execute for every 7top-level call of the interpreter. If we exceed this limit:

* We terminate execution
* The “results” left on the stacks can either be used or discarded

The “name” data type along with the “get” and “set” instructions allow us to store items of any type as a variable; this includes code for named subroutines ( which are much like functions in languages such as Python and C). If we encounter a symbol that is not either an instruction name or a constant, then we treat it as a “name” constant and push it to the “name” stack. Then this symbol can then be changed, using the “set” instruction, or read from, using the “get” instruction.

### 2.2.1 PushGP

PushGP is a Push implementation of Genetic Programming. It uses basic Genetic Programming Implementations. It randomly generates a population of initial programs in its initialisation. We find the fittest programs and select them for breeding through the Fitness evaluation phase. Then new individuals are created with mutation and crossover from the fittest-individuals. We select fittest individuals through tournament selection. Mutation picks a randomly chosen sub-expression and swaps it with another randomly chosen sub-expression. Crossover swaps one subexpression of an individual with the sub-expression of another.

### 2.2.2 Push3

Push3 is the newest version of Push in which many versions derivate from (such as pyshgp). The most significant change is the new “exec” stack which stores expressions, instructions and literals (reference). The values on the stack are values that are to be executed hence then name. It is independent of the code stack which we can still use for code manipulations. The exec stack holds the code queued for execution and is continuously executed upon. The basic process of executing a program (P) in push3 is as follows:

1. Push P onto the EXEC stack.

2. While the EXEC stack is not empty, pop and process the top element of the EXEC stack, E:

(a) If E is an instruction: execute E (accessing whatever stacks are required).

(b) If E is a literal: push E onto the appropriate stack. (c) If E is a list: push each element of E onto the EXEC stack, in reverse order.

(Lee Spector 2005) [9]

“quote” instruction now copies the top of the “exec” stack to the “code” stack. Previously, loops where done natively in languages that implement Push (such as python). Now it can be done by pushing a series of elements to the “exec” stack. When the loop is executed the elements on the “exec” stack are sequentially executed. Previously names have been bound to variables by the “get” and “set” instructions. Now, instead of using a quoted value on the “code” stack and using the “set” instruction, we instead use the “define” method on the “exec” stack. Executing subroutines is also more straightforward with them now being executed directly by calling just the name of the subroutine. Previously a “get” and “do” command would also accompany the name of the subroutine.

### 2.2.3 Linear Push (Plush)

In Plush, linear genomes called Plush genomes are used to represent programs. A genome is how we represent a program. These Plush Genomes are a linear set of instructions and literals which we translate to a push program before execution. Storing the push program like this allows us to perform genetic operations uniformly across the whole genome. Plush also uses “epigenetic” markers. It does this to help translate the Plush genome into a Push hierarchical structure for fitness testing. Inserting open parenthesis to indicate the start of code blocks and epigenetic markers to determine where the code blocks close. We insert the open parentheses when we call an instruction. Doing this allows for instructions to take code blocks from the “exec” rather than a single instruction.

#### Plush Genome to Push Program

We construct of the push program tree depth-first. We append each instruction and its arguments to the end of the Push Program. If the instruction has no arguments, it just appends the instruction. However, if they do have arguments, then the instruction is appended, followed by several code blocks corresponding to the number of arguments. For Example, an instruction which takes two arguments from the “exec” stack will be followed by two code blocks in the Push program. Inside these code blocks, there may be further instructions which we will perform the same process on, meaning we may have nested code blocks. The instructions indicate the opening of code blocks and the “:close” epigenetic marker indicates the closing of code blocks. The “:close” marker indicates how many open code blocks to close by way of a proceeding number. If this number were to exceed the number of open code blocks, then all code blocks are closed. If the end of the Plush genome is reached with code blocks still open, then these blocks are automatically closed.

### 2.2.4 PyshGP

PyshGP is a Python implementation of the push language. It gives a basic push instruction in Python. There are five standard data types that each have their own set of instructions. These data types are Boolean, Character, Code, Integer, Float and String. There is also typecasting present in PyshGP that allows conversion of one type to another. We do this by moving the relevant element from one stack to its new type stack.

An example would be the “integer\_to\_boolean” instruction that takes the top element of the integer stack and pushes a “False” to the Boolean stack if this integer is zero and “True”. Vector types also exist for some data types. These types also come with basic vector operations such as appendage, splitting and concatenation. Standard stack instructions exist for all stacks, these instruction include:

* dup: copy the top element in the stack
* swap: swaps the top two elements in the stack
* flush: remove all elements from the stack
* stack-depth: get the size of the stack and push this number to the integer stack
* yank: takes the integer from the top of the integer stack and uses this number as an index to find an element which we move to the top of the stack
* eq: compares the top two elements in the stack and pushes a “True” to the Boolean stack if they are equal and pushes a “False” otherwise

These are only a few of the instructions present in pyshGP. The other instruction would be more akin to operations seen in languages such as Java and Python.

It is possible to define new instructions if needs be. All new instructions are an instantiation of the Instruction class. This class must have a function, which takes in a PushState object and returns an altered PushState object. Every instruction uses the Instruction class and we store them in a set of registered instructions. Any new instruction must instantiate a new Instruction object and add it to this registered instruction set.

#### PyshGP Genetic Programming

In order to determine fitness, there are two ways. We could use a Fitness value which is a number out of one - hundred indicating how close to optimal a solution is (with zero being the least and one-hundred being the most). Alternatively, we can use an Error Value which is a positive number indicating how far away a solution is from optimal (with zero being the most optimal). In both methods, the user has to define either a “fitness function” or an error function to calculate the fitness value and error value, respectively. The pysgp system favours the “Error” approach. The error function defined must take in a push program along with a set of training cases. We calculate An error value for the program for each training case which we then return as an “error vector”. An individual will hold this “error vector” value along with the value of all the error values combined called the “total error”. This storage method allows the use of either in fitness selection operations.

Three fitness selection methods are supported. These are:

* Lexicase Selection
* Epsilon-Lexicase Selection
* Tournament Selection

We can represent our Push programs more linearly, doing so will allow us more flexibility in using genetic operators. We can achieve this by using Plush Genomes (see above). These Plush genomes can also be easily converted to Push programs for error evaluation. Uniform mutation is the only form of mutation currently supported in pyshgp. Additionally, the only recombination operator available in pyshgp is “alternation”. The user can define the parameters and tunable properties of these two genetic operators. Other values users can specify are the percentage use of each genetic operator to produce the next generation.

## 2.3 Evolutionary Robotics

Evolutionary Robotics is the area of study that involves evolving a suitable controller for a robot through some method of artificial evolution. To start, we must design a robot to perform a task. This robot consists of multiple layers where each layer is responsible for one behaviour. Usually, to coordinate necessary behaviours, we would use trial and error, this is infeasible as problems grow in complexity. Evolutionary Robotics is a suitable solution for this problem. Instead of using trial and error, we can use an Evolutionary Algorithm (such as Genetic programming) to find a suitable program. We do this through the basic EA process of defining a fitness function (or indeed error function )then finding gradually fitter and fitter control programs based on the robot’s interaction with the environment.

### 2.3.1 Applications in Genetic programming

There have been many different uses of Genetic Programming in evolutionary robotics. One such application was in 2003 when Krister Wolff and Peter Nordin created a robot that simulated bipedal motion on a robot through the use of Linear Genetic Programming [15]. Another use was the more maths heavy 2015 paper on the “synthesis of Mobile robots” by the four authors A.I. Diveev, S.I. Ibadulla, N.B. Konyrbaev and E.Yu. Shmalko [16].

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Type of Robot Used | Type of  EA used | Simulator Used | No. of Robots |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

Genetic Programming has been widely used in Evolutionary robotics as seen previously. However, one area which has had no research is a Push implementation of genetic programming for producing a mobile robot. This project aims to fix this.

## 2.4 Webots

Webots is a realistic platform for simulating mobile robots. It provides an environment to model, prototype and simulate robots in a variety of situations. It has real, commercially available robots in the simulations ready to be used for testing.

We can define many parameters for our robot such as shape, colour, texture, mass and friction.

Many different sensors are available for each of the robots to help it sense its environment such as distance sensors, range finders, light sensors and more.

We are also able to build an environment in which to simulate a robot. This environment cis customisable with different obstacles, different terrain and various other items.

Webots uses virtual time so that simulations can be run much faster (up to 300 times) than what would be possible in real life. The simulation can also be adjusted to suit a specific goal, such as accuracy or speed. Included in this simulation is a step-by-step which allows us a detailed view of how the robot behaves.

Webots is available for a multitude of languages such as the API’s for C and Python.

### 2.4.1 Application of Webots

Webots is a well-used platform in the world of Genetic Programming. It is used for many projects such as the 2014 paper by Pedro Silva, Christina P.Santos, Vitor Matos and Lino Costa [12]. In this paper, Webots is used as an evaluator for a controller implementing genetic programming which controls a biped locomotion robot. Another application is one of the primary papers of this project. The 2014 paper by [César Ferreira](https://ieeexplore.ieee.org/author/37085378197), [Pedro Silva](https://ieeexplore.ieee.org/author/37719072600), [João André](https://ieeexplore.ieee.org/author/38667548900), [Cristina P [11]. Santos](https://ieeexplore.ieee.org/author/37067256500)and [Lino Costa](https://ieeexplore.ieee.org/author/37332955200) creates a biped locomotive controller in Webots using Genetic programming. These are only a few of the many papers which show Webots effectiveness in this area

### 2.4.2 Other simulators and Webots

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Realistic Physics | Easy use of EA’s in environment | Customisable  Environment | Multitude of Sensors | Many different forms of robots | Is free |
| Webots | Yes | Yes | Yes | Yes | Yes | Yes |
| Gazebo | Yes | Fairly | Yes | Yes | Yes | Yes |
| Microsoft Robotics Deveeloper Studio | Yes | No | Fairly | Not a lot | Not a lot | Yes |
| Robot Virtual Worlds | Yes | No | Yes | No | Handful | No |
| V-REP | Yes | Fairly | Yes | Yes | YEs | Yes |

# 3 Requirements

|  |  |  |
| --- | --- | --- |
| Requirement | Functional/Non-Functional | MoSCoW |
| Robot Built-in Webots | Functional | Must |
| Flat environment built-in Webots with start and endpoint | Functional | Must |
| Robot has enough sensors to supply fitness function | Functional | Must |
| Initialisation method successfully implemented | Functional | Must |
| Fitness function implemented | Functional | Must |
| Fitness selection method implemented | Functional | Must |
| Genetic operators successfully implemented | Functional | Must |
| Robot can detect when it has reached the endpoint | Functional | Must |
| Have a complete Genetic program implementation in Push | Functional | Must |
| Steep Terrain added to Webots environment | Functional | Should |
| Implementation works on rough terrain | Functional | Should |
| Initialisation method ensures a diverse population | Non-Functional | Should |
| Genetic operators ensure diversity is maintained in the population | Non-Functional | Should |
| Simple block in the way of path implemented in Webots | Functional | Could |
| Implementation works at finding path around block | Functional | Could |
| Various obstacles implemented in Webots Environment | Functional | Would |
| Implementation works for all obstacles | Functional | Would |

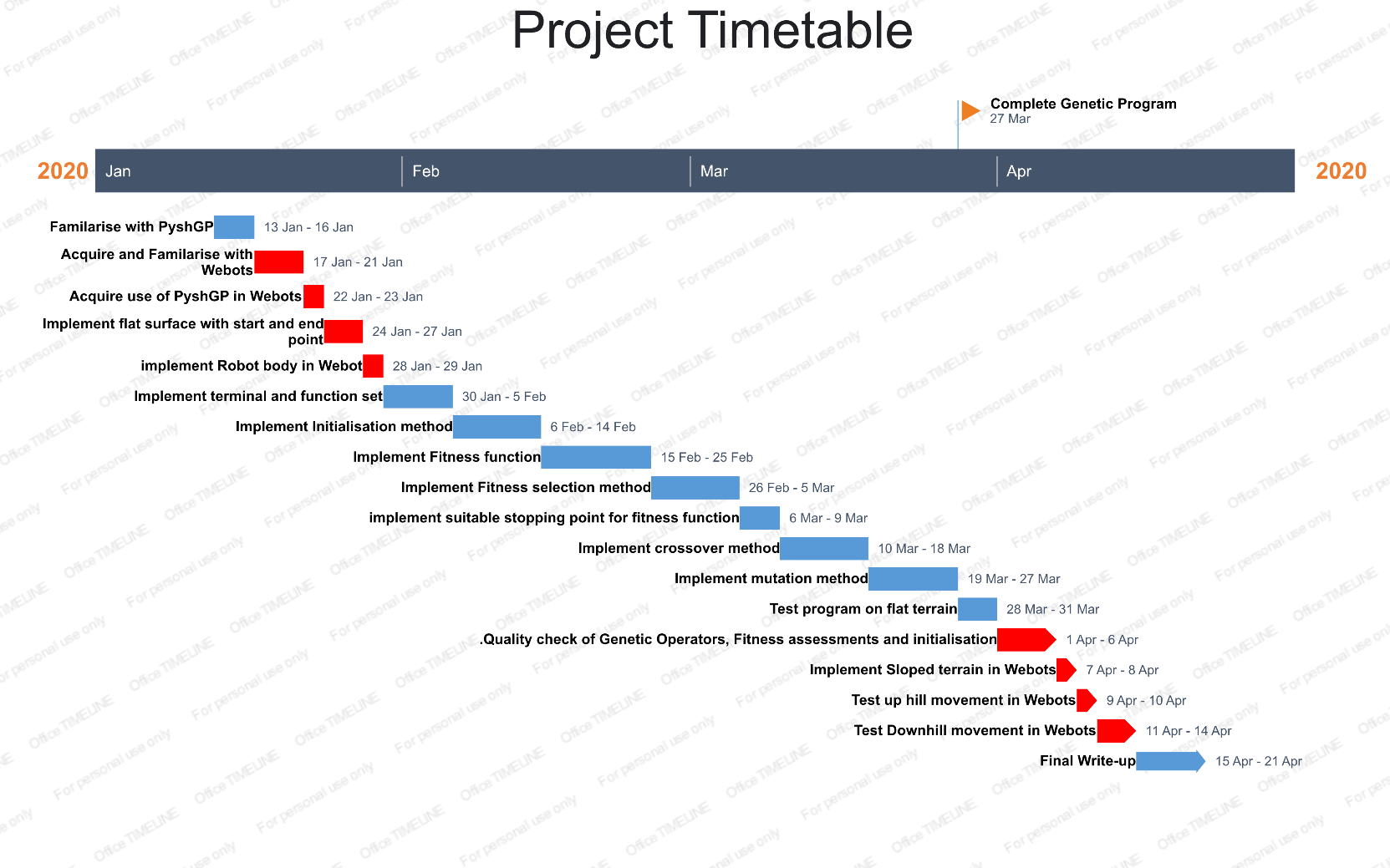
# 4 Design

The design for the robot, its fitness functions, its initialisation, its genetic operators and its testing environment will be taken from the 2014 paper on Genetic programming applied to biped locomotion. We will implement this in the Push programming language, more specifically the Python implementation of it (pyshgp). As in the paper, the environment I use will be Webots. However, if this proves to be too complicated to implement, then we will decide to create a new fitness function, initialisation method, genetic operators etc. The difference is that it will be much simpler and possibly less effective but more realistic to implement

# 5 Evaluation Strategy

# 6 Project Management Plan

## 6.1 Timetable



## 6.2 Risk Analysis

|  |  |  |
| --- | --- | --- |
| Risk | Likelihood/Impact | Mitigation |
| Task takes longer than expected | High/Medium | Meet with supervisor and discuss what to prioritise |
| Implemented GP can not find a path to goal | Low/Medium | Readjust parameters and re-run tests or change some algorithms in the GP |
| GP is too complicated to implement | Medium/Medium | Write simpler GP |
| Unable to have a successful moving robot | Low/Medium | Change body used or look to fix the current body |
| Unable to get PyshGP working in Webots | Low/Medium | Change to another version of Push or choose another testing environment |
|  |  |  |

# 7 References

[1] Poli, R., Langdon, W., McPhee, N. & Koza, J., 2008. *A field guide to genetic programming*. [S.l.]: Lulu Press.

[2] Luke, S. (2015). Essentials of Metaheuristics.

[3] Spector, L., 2012. Assessment of problem modality by differential performance of lexicase selection in genetic programming. *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference companion - GECCO Companion 12*.

[4] Cava, W.L., Spector, L. & Danai, K., 2016. Epsilon-Lexicase Selection for Regression. *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference - GECCO 16*.

[5] Spector, L., 2001. Autoconstructive Evolution: Push, PushGP, and Pushpop Proceedings of the Genetic and Evolutionary Computation Conference

[6] Pantridge, E. & Spector, L., 2017. PyshGP. *Proceedings of the Genetic and Evolutionary Computation Conference Companion on - GECCO 17*.

[7] Brameier, M., 2005. On Linear Genetic Programming. (maybe extra and another citation here)

[8] Spector, L., 2003. *Push 3.0 Programming Language Description*. Available at: http://faculty.hampshire.edu/lspector/push3-description.html [Accessed November 13, 2019].

[9] Spector, L., Klein, J. & Keijzer, M., 2005. The Push3 execution stack and the evolution of control. *Proceedings of the 2005 conference on Genetic and evolutionary computation - GECCO 05*.

[10] Pratihar, D., 2003. Evolutionary Robotics. *SpringerReference*.

[11] Ferreira, C. et al., 2014. Genetic Programming Applied to Biped Locomotion Control with Sensory Information. *Proceedings of the 11th International Conference on Informatics in Control, Automation and Robotics*.

[12] Silva, P. et al., 2014. Automatic generation of biped locomotion controllers using genetic programming. *Robotics and Autonomous Systems*, 62(10), pp.1531–1548.

[13] Olivier, M., Cyberbotics Ltd. Webots™: Professional Mobile Robot. Available at: https://journals.sagepub.com/doi/full/10.5772/5618 [Accessed November 13, 2019].

[14] Hohl, L. et al., 2006. Aibo and Webots: Simulation, wireless remote control and controller transfer. *Robotics and Autonomous Systems*, 54(6), pp.472–485.

[15] Wolff, K. & Nordin, P., 2003. Learning Biped Locomotion from First Principles on a Simulated Humanoid Robot Using Linear Genetic Programming. *Genetic and Evolutionary Computation — GECCO 2003 Lecture Notes in Computer Science*, pp.495–506.

[16] Diveev, A. et al., 2015. Variational Genetic Programming for Optimal Control System Synthesis of Mobile Robots. *IFAC-PapersOnLine*, 48(19), pp.106–111.

[17] Baeck, T., Fogel, D., Michalewicz, Z. 2017. *EVOLUTIONARY COMPUTATION 1: basic algorithms and operators*, Place of publication not identified: CRC Press.