

Genetic Algorithms for Robot Mobility

Deliverable 1: Final Year Dissertation

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

# Introduction

## AiM

To implement a Genetic Algorithm in a robot for it to learn how to move in a simulated environment to show how Robots learning through Genetic Algorithms would perform in the real world.

## objectives

* Use Webots to Create a suitable simulated environment for the robot to move in ( M )
* Create a body for the robot to operate in Webots ( M )
* Add alternate bodies for the robot to operate in Webots( C )
* Create a genetic algorithm that allows a robot to traverse flat ground over a short distance ( M )
* Create a genetic algorithm for a robot to traverse uneven ground ( S )
* Create a genetic algorithm that allows a robot traverse obstacles ( C )
* Create a genetic algorithm that allows a robot to jump over gaps ( W )
* Apply different approaches to GA’s throughout the different scenarios ( C )
* Optimize the algorithm to move the robot efficiently across the terrain ( W )
* Have the robot learn the path from scratch as quickly as possible ( W )
* Have the robot learn to navigate a multitude of terrains at once ( C )

## 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

# Background

## Evolutionary Algorithms

Evolutionary Algorithms (EA) take influence from nature and biology to try and solve problems. 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
* Steady-state algorithms: meaning they choose a subset of individuals to iterate over at a time

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:

1. Initialization of the population
2. Fitness Assessment to find the current most optimal solution
3. 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 initialize the population, how to assess their fitness, and how to do crossover and mutation.

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.

### 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.

#### Initialization

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. For example, we can represent GP’s as trees or lists, and a GA can be represented using vectors of Boolean values, real values or integers.

For example, there is Fitness Proportionate Selection can work but has some definite drawbacks.

#### 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. Meaning that this crossover will struggle to find a solution that requires a secure connection between the first and last vector or one that requires that we split first two t split up often

It’s important to note that crossover can’t get every conceivable solution as it only produces children which preside in a ‘box’ which can’t ever increase. To counter this and make sure we aren’t stuck in this box we do mutation.

#### 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. This is a very simple mutation that can work quite well but the drawback is that it only works on binary vectors.

### Genetic Programming

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

#### Linear GP

GPs can be represented as arbitrary length lists of machine language instructions. It is important that closure is maintained while executing arbitrary machine code strings. To maintain closure, it is important to construct your list from a carefully selected set. For Example: If instructions are finite in length then you can assign a unique integer to each instruction. Usually uses a finite set of registers, meaning that the machine code lists operate more like directed acyclic graphs in that 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 are normally initialized 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 important 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 normally used to represent imperative or machine languages.

##### Initialization

Lists are generated per the constraints of the domain they are required for. However, there are 2 general issues when initializing a list: the length of the list and populating it. Usually Initialization is done completely randomly. 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 important 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.

##### Fitness Assessment

The way you select your fitness assessment can have very large consequences on what solutions you put out. Fitness assessment is the way in which fit individuals are selected from a population and is usually done in two stages:

1. Fitness Evaluation: Each individual’s fitness is assessed based, usually, on a fitness function which takes into account multiple variable to sense how “well” a program has done at its given task.
2. Fitness Selection: Process in which individuals are picked for Crossover and mutation. Usually fitter individuals are more likely to be picked.

###### Tournament Selection

Tournament selection is one of the most popular methods for fitness selection in many GA’s and even GP’s. There are a number of reasons for this such as: it’s not very sensitive to how the actual fitness function works, it’s very simple, requires no preprocessing, works well with parallel algorithms (important for running as many iterations as possible at once), 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 and 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. This can be done 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.

##### Crossover

Crossover is considered a macro genetic operation. Crossover is usually done via the “two-point linear crossover” method

###### Two-Point Linear Crossover

Two-point linear crossover is done by taking a parent, selecting two points between zero and the length of the parent then taking the other parent and doing the same then swapping the values inside these two ranges. There is a problem that comes from this in 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

Mutation is done to keep variation in the population. It is important to keep your population diverse in order to stop your algorithm prematurely coming to a halt when it thinks it has found the optimal solution, but this solution may be a local optima and not the most optimal solution for the problem (global optima). Although mutation is not as common in GP as it is in other EA’s it is still possible and is often recommended in some cases (as with One-point Crossover) to prevent premature conversion.

\*insert graph showing local and global optima

Mutation is considered a micro genetic operation. Mutation can be done 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 only one genetic operator is used, then it is likely that the variation won’t be high, and the evolution will progress more precisely with smaller changes throughout each iteration and would be more exploitative. However, if multiple genetic operators are used at once then it will make much bigger 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) Some of the nodes of a tree representations may be restricted to how many children they can have, this is an important quality for the construction of trees in initialization.

##### Initialization

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 these rules must be followed.

One common algorithm for initialization is the Grow algorithm. In the grow algorithm a tree is built 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” doesn’t.

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 special node called the Ephemeral Random Constant (ERC). When an ERC is picked it is transformed into a randomly generated constant of your choosing. Once this has happened the value of the ERC can not be changed except through special mutator methods.

##### Crossover

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 and we hope to combine them to get a good trait from both parents to create an even fitter child.

In GP, the 3 most basic ways of doing 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 sub trees. It is Normally 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 (essentially 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.

On of the important properties of the One-Point Crossover method is “it makes the calculations necessary to model the distribution of GP schemata feasible” meaning that 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 it is important that we mutate.

##### Mutation

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 max depth of 5 is used for the generation of the subtree. This 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

A random mutation point is chosen in the Tree. Take the value of this node and replace it with a value (that has the same arity as the selected node) from the function set if it is a branch node or from the terminal set if it is a leaf node with the same arity e.g. 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 where as point mutation is done on a per node basis.

##### Automatically Defined Functions

Trees can be used to define automatically defined functions which can be called be a primary tree. ADF’s can search a large space if an optimal solution is known to be repetitive, . It can also make it easier on a individual by breaking individuals into modules.

## Push

Designed to have a completely uniform syntax so the ease the process of generating code that manipulates code (say for auto constructive evolution), all while supporting a multitude of different data types as well as recursion and sub-routines. 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: the brackets must be balanced. 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. Its 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 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.

Some instruction (such as ‘+’) may be skipped 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. This is 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 the ‘quote’ instruction is executed, the next piece of code to go through execution will be pushed to the expression 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 top-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.

### 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.

### Linear Push (Plush)

In Plush, programs are represented by linear genomes called Plush genomes. A genome is how a program is represented. These Plush Genomes are a linear set of instructions and literals which is translated 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 in order to help translate the Plush genome into a Push hierarchical structure for fitness testing. This is done by inserting open parenthesis to indicate the start of code blocks and epigenetic markers to determine where the code block is closed. They are inserted when an instruction is called. This allows for instructions to take code blocks from the “exec” rather than a single instruction.

#### Plush Genome to Push Program

Construction of the push program tree is done depth-first. Each instruction and its arguments are appended to the end of the Push Program. If the instruction has no arguments it is just appends the instruction. However, if they do have arguments then the instruction is appended followed by a number of code blocks corresponding to the number of arguments. For Example: an instruction which takes 2 arguments from the “exec” stack will be followed by 2 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 where to close the code blocks. The “:close” marker indicates how many open code blocks to close by way of a proceeding number . If this number where 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.

### 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. Common 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 is then moved 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 a kin to operations seen in languages such as Java and Python.

It is possible to define your own instruction if needs be. All new instruction 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 are stored 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 2 ways. You can either 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, you 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 a long with a set of training cases. A error value is calculated for the program for each training case which is then returned as an “error vector”. An individual will hold this “error vector” value a long with the value of all the error values combined called the “total error”. This 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 that is currently supported in pyshgp. As well as the this, the only recombination operator is alternation. The parameters and tunable properties of these two genetic operators can be defined by the user

# Project Management Plan

Research the multiple different implementations of genetic algorithms

## Work Breakdown Structure

### 1 –Initial Set-up and Research

* 1. - Research multiple different Genetic Algorithm implementations
  2. - Pick one Implementation method for GA’s to start with initially

1.3 - Acquire the use of Webots and learn basics

1.4 - Create first environment with flat ground in Webots

1.5 - Create Initial robot body in Webots

1.6 - Add a start position and goal for Robot

### – Flat Ground Scenario Genetic Algorithm

* 1. – Implement first Genetic Algorithm for flat ground traversal
  2. – Combine the algorithm with the Webots Robot
  3. - Run GA in the Flat Ground scenario until Robot reaches goal or time runs out
  4. - Modify GA until Robot reaches goal or try different method for GA
  5. – Continue to run GA until Robot converges on method to reach goal

### 3 – Uneven Ground Scenario Genetic Algorithm

3.1 - Change Webots environment to include just uneven terrain on path

3.2 – Run Flat Ground GA on Uneven ground until robot reaches goal(Risk: or time runs out)

3.3 – Modify Flat Ground GA until Robot Reaches goal (Risk: or try different method for GA)

3.4 – Continue to run GA until robot converges on method to reach goal

3.5 – Add in flat ground and Uneven ground to simulation

3.6 – Run same GA on the new terrain until robot reaches goal

3.7 - Continue to run GA until Robot converges on method to reach goal

### 4 – Obstacle Scenario Genetic Algorithm

4.1 – Change Environment so there is a single obstacle on path to goal

4.2 – Run Uneven ground GA on Obstacle path until robot reaches goal

4.3 – Continue running GA until Robot converges on method to reach goal

4.4 – Add in flat and uneven ground to simulation before obstacle

4.5 – Run Same GA on new terrain until robot reaches goal

4.6 – Continue to run GA until robot converges on method to reach goal

### 5 – Final Testing of Genetic Algorithm

5.1 – Take final algorithm and run it on flat ground

5.2 -

### Webots

* Implement Robot in Webots
* In depth learning of the functionality and various uses of Webots
* Create alternate bodies in Webots

### Implementing Genetic Algorithms

* Implement first Genetic Algorithm in robot for traversing flat ground
* Use Flat ground GA for uneven ground and modify accordingly
* Use first ground GA for

### Testing

* Test Robot on flat surface
* Track how long Robot takes to learn path for flat ground
* Track how long Robot takes to learn path for uneven ground
* Track how long Robot takes to learn path for Obstacles
* Track how long Robot takes to learn path when robot has to jump over a gap