

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 for the purpose of showing 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 a 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) or are Steady-state algorithms (meaning they choose a sub set 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 they are formed. Both can either be generational or steady-state but both generally: create a set of candidate solutions which are tested, the best solutions are then used to derive further solutions (may be generational or Steady-state). This cycle is then repeated until either a suitable solution is found or we run out of time. The methods described above have names and are a part in 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 a solution is 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 some steps completely. I will touch on Genetic Algorithms but my focus here is on Genetic Programming.

### Genetic Algorithms

The first idea of genetic algorithms was conceived by John Holland at the University of Michigan in 1970. This GA selects individual’s 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 biggest decision is how you represent your population. There are a few different representations depending on what EA you’re are using, for example: GP’s can be represented as trees or lists, a GA can be represented using vectors of Boolean values, real values or integers.

#### Fitness Assessment

The way you select your fitness assessment can have very large consequences on what solutions you put out.

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

##### Fitness Proportionate Selection (Roulette Selection)

Select individuals in proportion to their fitness. The higher the fitness the more likely that individual is to be picked. A ‘Size’ is given to an individual based on their fitness so a higher fitness means a bigger ‘Size’. Then a number is picked between 0 and the total size of all the individuals. This number is essentially a pointer like on a roulette wheel but the roulette wheel is made up of all the individuals and the fitter individuals have a bigger slice of the wheel, as such if the number lands within an individual; that individual is picked.

\*insert pick of roulette selection

One of the advantages is that it is more likely that a fitter individuals is picked as well as still allowing less fit individuals to be picked which may seem counterproductive but it actually helps keep our population which is important (as discussed in the mutation section). However this method also has the disadvantage of struggling to pick the absolute optimum e.g. for values between 0 and 1 it would struggle to pick the more optimum o.999 from the less optimum 0.998 or 0.997.

One of the way we can fix this is to use a non-parametric fitness selection algorithm where in fitness is only measured by “bigger is better”. One method that does this is Tournament selection

##### 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 each individual 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

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.

There are 3 basic ways of doing crossover:

* One-Point Crossover
* Two-Point Crossover
* Uniform 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 2 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 strong connection between the first and last vector or one that requires the first two to be split up often

##### Two-Point

This method attempts to overcome some of the problems of the previous. Start by selecting 2 points between 1 and the length. Then swap the indexes between these points. Crossover still might not happen if the two numbers picked are the same. There is still a linkage problem similar to One-point Crossover in that items further away from each other are more likely to be broken up.

##### Uniform Crossover

This beats the previous mentioned problems by crossing over each point independently of one another. This having a random probability of each point to be crossed over. This is the superior version of the 3 basic crossover as it doesn’t have the linkage problems as the other 2.

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

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) .

\*insert graph showing local and global optima

An example of Mutation would be 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. 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 optimizing small computer programming. 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. Some of the nodes of a tree representations may be restricted to how many children they can have.

#### Tree Style Genetic Programing Pipeline

Tree style GP is the most common form of genetic programming. It, as the name implies, uses trees as its representation. Usually GP are constructed like GA’s except their crossover and mutation is different in that it does crossover with 2 parents 90% of the time and 10% of the time the parent is directly copied. There isn’t usually mutation as the strange crossover method is usually enough . However mutation is still possible.

## 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 basically 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 reading it for a human might be a little difficult (more on this later). Some instruction (such as ‘+’) may be skipped if there aren’t 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 really there for structure. Push also allows for code manipulation through its ‘code’ data type which inherits from the ‘expression’ data type. An important 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 actually being executed.

# 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 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