Genetic programming for building voxel structures with turtle robots

Iacobescu Tudor

2021

Chapter 1

Introduction

This thesis will explore the construction of three-dimensional voxel structures using a "turtle"-type virtual robot. This robot is programmable, with the ability to interact with its surroundings by means of a set of elementary functions. These functions allow it to move, place and destroy blocks, and detect adjacent blocks for the purpose of making decisions.

This project initially took form based on an existing concept: a modification for Minecraft that adds such a robot to the game, which is programmable using Lua. Minecraft is a sandbox video game where players can build structures using 3D blocks aligned to a grid. The modification, *ComputerCraft*[3], adds simple Lua-programmable computers to the game. Among them is a mobile computer, called a *turtle*, that can use a special library to interact with its environment.

The idea that the thesis is built around is using genetic programming to create, given a target voxel structure, a program that can construct it. When run by a turtle with sufficient materials, this program should allow it to reproduce the initial target as accurately as possible.

Chapter 2

Practical context

In this chapter, I will outline the precise problem that needs to be solved, as well as the constraints of the work environment.

2.1 The turtle

The turtle is a robot that takes up one space in the voxel grid. Its state is described using a triplet of integer coordinates, as well as a direction aligned to one of the horizontal axes.

For the precise abilities of the robot, I will take inspiration from ComputerCraft, the previously mentioned modification. From the library of commands available for turtle control in ComputerCraft, I will utilize only the following:

- The commands `forward`, `back`, `up` and `down` allow the robot to move to an adjacent space. If said space is taken up by a block, the robot will remain in place. These commands will return a boolean value, equal to the success of the movement.

- The commands `turnLeft` and `turnRight` allow the turtle to turn 90 degrees clockwise or counter-clockwise. They cannot be blocked, and will not return anything.

- The `place`, `placeUp` and `placeDown` commands allow blocks to be placed in adjacent spaces. If the spaces already have a block, then the placement will fail, and the commands will return a boolean `false`; otherwise, `true` will be returned.

- The `dig`, `digUp` and `digDown` commands allow erasing blocks from adjacent spaces. If the positions are already free, or are outside the training space, the commands will fail. They will return a boolean value, equal to the success of the operation.
- `detect`, `detectUp` and `detectDown` will allow detecting blocks next to the turtle. They will return `true` if a block is present in the given space, and `false` otherwise.

One interesting thing is that most of the commands only allow interactions directly in front of, above and below the turtle (`back` also allows moving backwards). This means that to move into or interact with the positions on the sides of the turtle, it must first turn to face them. Another important consideration is that the operations that place or destroy blocks only affect adjacent spaces, and thus the robot needs to move to a location in order to interact with it.

2.2 The training space

In Minecraft, a turtle can interact with most blocks in the full 3D space, which is effectively unbounded on the horizontal axes. For my purposes, I will use a limited 3D space, only 16 blocks across in every direction. The robot can move only inside the training space, and only affect voxels contained within it. The area out-of-bounds can be considered, for the purposes of turtle logic, to be made up of unbreakable blocks.

Every position in the space can be identified by a triplet of coordinates (x, y, z) where $x, y, z = \overline{0..15}$ and where the y-axis is vertical and the x- and z-axes are horizontal. This is analogous to the inspiration material, as Minecraft also uses *y-up* space. The positions can either be free, or taken up by a block (or voxel). The initial position of the turtle in the training space is $(0, 0, 0)$, and its direction is towards positive x. Every position is initially free, and may be filled by the turtle during its operation.

Chapter 3

Technical details

The idea of the thesis is to attempt the generation of programs for turtles evolutionarily, using a genetic algorithm. Ideally, these programs should also be reasonably easy to translate from the abstraction used in the training algorithm to Lua code that can actually be run on a Minecraft turtle. However, this presents a problem: genetic programming works more naturally with programs structured as trees, but Lua is not a functional language, and thus does not translate easily into trees. This means that an additional layer is required, leading to the following program structure:

- a simulator for the turtle and the voxel space that can easily be controlled by outside code
- a virtual machine that can run linear programs (that can easily be translated to imperative code) that can interact with an array of registers and with the simulator
- a higher level tree-based structure for programs that can easily be mutated and mixed together
- a genetic algorithm that works with populations of tree-based programs, evolving them over time

This structure is implemented in Rust, a modern multi-paradigm programming language presenting several advantages, including:

- high-level language features with a low overhead

- static typing, leading to higher performance
- a memory allocation system that is safe without the need for a garbage collector, leading to easy parallelization

Rust's memory safety, which prevents many hard to track down bugs, has lead to it being used alongside C and C++ in projects such as the Linux kernel. High performance and concurrency are required for a genetic algorithm as well, so I felt that Rust was a good fit for this project.

3.1 The simulator

The simulator for the training space and the turtle itself is fairly simple. Its structure is as follows:

```rust
pub struct Simulator {
    blocks: BlockSpace, // 3D array of bytes
    turtle: Turtle
}
```

The blocks are stored in a 3D array.

```rust
pub(crate) const BLOCK_SPACE_SIZE: usize = 16;
pub type BlockSpace =
    [[[u8; BLOCK_SPACE_SIZE]; BLOCK_SPACE_SIZE]; BLOCK_SPACE_SIZE];
```

The turtle has a position (a triple of coordinates) and a facing direction (on either of the horizontal axes, towards plus or minus).

```rust
struct Vector3(isize, isize, isize);
enum Orientation {
    XPos,
    XNeg,
    ZPos,
    ZNeg
}

struct Turtle {
    pos: Vector3,
    facing: Orientation
}
```

Commands to the turtle are called as methods on the `Simulator` instance. These methods will modify the data of the turtle and the blocks as needed. For example, the following is the code for attempting to move the turtle in a direction (forwards, up or down):

```
pub fn try_move(&mut self, dir: Direction) -> bool {
    let pos = self.turtle.get_adjacent(dir);
    if Self::pos_in_bounds(pos) {
        let Vector3(x, y, z) = pos;
        if self.blocks[x as usize][y as usize][z as usize] == 0 {
            self.turtle.shift(pos);
            true
        } else {
            false
        }
    } else {
        false
    }
}
```

3.2 Linear programs and the virtual machine

Linear programs are encapsulated with the state of the virtual machine that runs them as a single entity. This leads to the following structure:

```
pub struct Program {
    instructions: Vec<Instruction>,
    labels: HashMap<Label, Ins>,
    registers: [Val; 256],
    instruction_pointer: Ins,
    halted: bool,
    simulator: Simulator,
}
```

The member variable `instructions` contains the actual "code" of the program, while the rest represent either information derived from it (`labels`, used for evaluating jump instructions) or the current state of the VM. In detail, these are:

- `registers`, which are variables that can be changed by instructions in the code

- `instruction_pointer`, which stores the index of the currently executed instruction in the instruction array
- `halted`, a boolean value which becomes `true` when execution has finished
- the `simulator`, which can be interacted with using certain “turtle” instructions

Each instruction in `instructions` is one of several types:

```
pub enum Instruction {
    /// Do operation on register alone
    Unary(Reg, UnaryOperation),
    /// Do operation on register using source
    Binary(Reg, Source, BinaryOperation),
    /// Jump to the given instruction if it exists, and if the condition is met
    Jump (Label, JumpCondition),
    /// Compare register with source, store value in comparison register
    Compare (Reg, Source),
    /// Print the source value
    Print (Source),
    /// Execute turtle operations in the simulator
    Turtle (TurtleOperation),
    /// Label for jumping
    Label (Label),
    /// Do nothing
    Pass
}
```

The `UnaryOperation` type holds unary operations such as incrementing or shifting, while `BinaryOperation` holds binary operations such as addition or the binary “AND” operation. `TurtleOperation`, on the other hand, holds operations specific to the problem domain, as described in section 2.1:

```
pub enum TurtleOperation {
    Move(Direction),
    Turn(Side),
    Place(Direction),
    Dig(Direction),
    Detect(Direction),
}
```

To execute the code, the VM goes through instructions step by step, processing each one. For example, the following code snippet handles the `Unary` instruction type:

```
Instruction::Unary(reg, op) => {
    let old = self.get_reg(reg);
    let new =match op {
        UnaryOperation::Not => !old,
        UnaryOperation::ShiftLeft => {
            let lost_bit = old & (1 << 7);
            self.set_reg(RESULT_REGISTER, lost_bit);
            old << 1
        },
        UnaryOperation::ShiftRight => {
            let lost_bit = old & 1;
            self.set_reg(RESULT_REGISTER, lost_bit);
            old >> 1
        },
        UnaryOperation::RotateLeft => i8::rotate_left(old, 1),
        UnaryOperation::RotateRight => i8::rotate_right(old, 1),
        UnaryOperation::Increment => i8::overflowing_add(old, 1).0,
        UnaryOperation::Decrement => i8::overflowing_sub(old, 1).0,
    };
    self.set_reg(reg, new);
}
```

One can notice here the use of a special register, `RESULT_REGISTER`. It and the `COMPARE_REGISTER`, are set by some operations and can be read like any other register. This allows programs to read and use, for example, the result of turtle operations, bits lost by the bit shift operation, or the results of comparisons.

```
Instruction::Compare(reg, src) => {
    let other = self.get_source(src);
    let this = self.get_reg(reg);
    self.set_reg(COMPARE_REGISTER, match i8::cmp(&this, &other) {
        Ordering::Less => -1,
        Ordering::Equal => 0,
        Ordering::Greater => 1,
    });
}
```

While for most instructions the instruction pointer will just be incremented by one each time, the jump instructions can use labels to alter program flow conditionally:

```
Instruction::Jump(label, condition) => {
    let do_jump = match condition {
        JumpCondition::None => true,
        JumpCondition::Zero(reg) => self.get_reg(reg) == 0,
        JumpCondition::NotZero(reg) => self.get_reg(reg) != 0,
        JumpCondition::Compare(ord) =>
            ord_to_num(ord) == self.get_reg(COMPARE_REGISTER),
        JumpCondition::NotCompare(ord) =>
            ord_to_num(ord) != self.get_reg(COMPARE_REGISTER),
    };
    if do_jump {
        let destination = self.labels.get(&label);
        if destination == None {
            return Err("Attempted to jump to missing label");
        }
        let destination = destination.unwrap();
        self.instruction_pointer = *destination;
        jumped = true;
    }
},
```

Lastly, turtle operations interact with the simulator and place their results in the RESULT_REGISTER.

```
Instruction::Turtle(op) => {
    match op {
        TurtleOperation::Move(dir) => {
            let res = self.simulator.try_move(dir) as i8;
            self.set_reg(RESULT_REGISTER, res);
        },
        TurtleOperation::Turn(side) => self.simulator.turn(side),
        TurtleOperation::Place(dir) => {
            let res = self.simulator.try_place(dir) as i8;
            self.set_reg(RESULT_REGISTER, res);
        },
        TurtleOperation::Dig(dir) => {
            let res = self.simulator.try_dig(dir) as i8;
```

```
                self.set_reg(RESULT_REGISTER, res);
            },
            TurtleOperation::Detect(dir) => {
                let res = self.simulator.detect(dir) as i8;
                self.set_reg(RESULT_REGISTER, res);
            },
        }
    }
```

3.3 Tree-based programs

3.3.1 Motivation

The virtual machine can run fairly complicated programs, but linear programs are not a good fit for genetic programming. A genetic algorithm needs to be able to mutate individuals and splice them together, and linear programs are too sensitive to change for this to work well. A perfectly functional linear program can be easily be completely broken by changing a single instruction.

In fact, using linear programs directly was something I initially attempted; I would translate an instruction into one or more bytes, effectively turning a set of instructions into a byte array. This, however, made the programs even more sensitive to change, as modifying a single byte belonging to a multi-byte instruction could make it so that the byte array could not even be translated into working code anymore. Even if the bytes could be translated into code, there was no guarantee that a single-byte mutation would not radically change the functionality of the program.

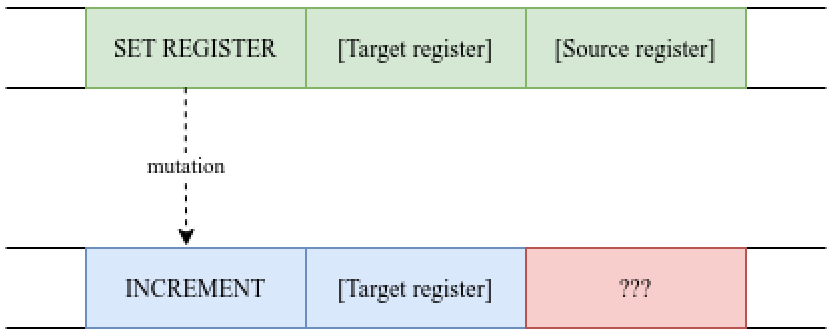

Figure 3.1: Mutating the first byte of a three-byte instruction can turn it into a two-byte instruction, leaving one byte to change the meaning of bytes after it.

Using linear programs for genetic programming is not impossible, and it has been done to great success in the past. However, the extra complexity that this would introduce was something I wanted to avoid. As such, I opted to instead use tree-based programs, something which is more natural to functional programming than imperative programming. Still, converting a tree to an imperative program is not difficult, and trees would be much easier to work with for the genetic algorithm.

3.3.2 Structure

The tree-based programs are constructed using nodes, which can have a function or value and, depending on their arity, a number of children. The types of nodes implemented are as follows:

```
pub enum Node {
    Null,
    Val(Source),
    Unary(UnaryOperation, Box<Node>),
    Binary(BinaryOperation, Box<Node>, Box<Node>),
    Then(Box<Node>, Box<Node>),
    Print(Box<Node>),
    Store(Reg, Box<Node>),
    /// condition, if_not_zero, if_zero
    If(Box<Node>, Box<Node>, Box<Node>),
    /// condition, block; repeat block until condition equals zero
    While(Box<Node>, Box<Node>),
    /// count, block; repeat block count times
    Repeat(Box<Node>, Box<Node>),
    Compare(Box<Node>, Box<Node>),
    Turtle(TurtleOperation),
}
```

I will also be representing these trees graphically, to aid in their understanding for the purposes of this thesis. For example, the following literal is equivalent to the image below it:

```
Binary(
    BinaryOperation::Add,
    Box::from(Binary(
        BinaryOperation::Add,
        Box::from(Val(Source::Value(1))),
        Box::from(Val(Source::Value(2))),
    )),
    Box::from(Binary(
        BinaryOperation::Multiply,
        Box::from(Val(Source::Value(3))),
        Box::from(Val(Source::Value(4))),
    )),
)
```

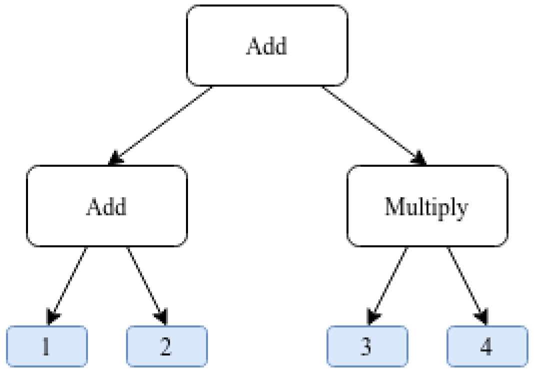

Figure 3.2: A simple tree-based program

Since the purpose of these programs is to build structures, they must allow for side effects. To this purpose, the full instruction set is as follows:

- `Null`: do nothing.
- `Val`: supply a value, either from a register or directly.
- `Unary` and `Binary`: execute an operation, using its children as parameters.
- `Then`: execute the left child, followed by the right one.
- `Print`: print the value of its child returning it, used for debugging.
- `Store`: store the value of its child in the given register.
- `If`: depending on the value of its leftmost child, execute and return the value of either its middle child or the right one.
- `While`: while its left child has a value different from 0, execute the right one, returning its value on the last iteration.

- `Repeat`: repeat its right child as many times as the value of the left child.
- `Compare`: compare the left and right children, returning the comparison result.
- `Turtle`: execute a turtle operation and return its result.

When mutating a tree is required, it is possible to change out a node for one of the same arity, or even for a completely new subtree, and still have a functioning program.

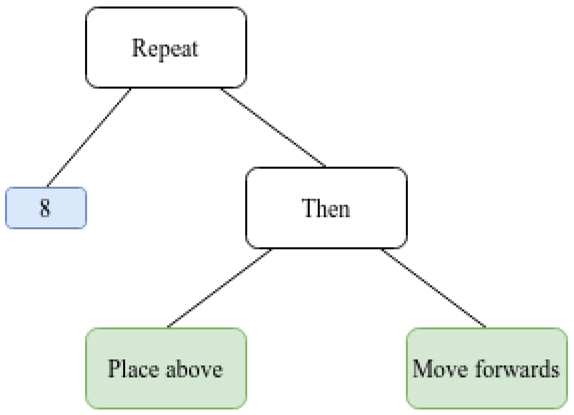

Figure 3.3: A simple program that builds a line from $(0, 1, 0)$ to $(7, 1, 0)$.

3.3.3 Translation

The end-goal, as before, is to generate a linear program. This means that a mechanism to convert a tree-based program into a linear one is required.

There are some subtleties to be aware of: the trees are not entirely functional (side effects complicate things), and their code has to be translated into the linear structured described before. The mechanism chosen for this is by making use of a *stack pointer*. This is a variable that holds a reference to a certain register during the translation process, starting with `STACK_START` (100). This variable is incremented when recursing deeper into the tree, and is used to identify the register that should be used for storing the result returned by a subtree. Henceforth, the register referred to by this pointer shall be denoted `r[stack_ptr]`.

Additionally, *labels* are used for control-flow structures. A label is a (linear) instruction holding an unsigned 16-bit integer, used by jump instructions as a target. Every new label instruction receives a new integer, guaranteeing their uniqueness.

Translation follows the following mechanism, where `translate_subtree` is a recursive function:

```
pub fn translate_tree(tree: &Node) -> Vec<Instruction> {
    let mut next_label = 0u16;
    translate_subtree(tree, STACK_START, &mut next_label)
}

fn translate_subtree(tree: &Node, stack_ptr: u8, next_label: &mut u16) -> Vec<Instruction> {
    match tree {
        ...
    }
}
```

For example, a subtree rooted in a unary operation is translated from its children first. Afterwards, the operation is executed on `r[stack_ptr]`, which will have been set to the result of the child subtree.

```
Node::Unary(op, child) => {
    // do the subtree, which stores result in r[stack_ptr]
    let mut instr = translate_subtree(child, stack_ptr, next_label);
    // do op on r[stack_ptr]
    instr.push(Instruction::Unary(stack_ptr, *op));

    instr
}
```

For multiple subtrees, the stack pointer is incremented, so that two separate values may be used:

```
Node::Binary(op, left, right) => {
    // do left subtree, which stores result in r[stack_ptr]
    let mut instr = translate_subtree(left, stack_ptr, next_label);
    // do right subtree, which stores result in r[stack_ptr + 1]
    instr.append(&mut translate_subtree(right, stack_ptr + 1, next_label));
    // do op on r[stack_ptr] with r[stack_ptr + 1]
    instr.push(Instruction::Binary(stack_ptr, Source::Register(stack_ptr + 1), *op));

    instr
}
```

A `Then` node simply adds the instructions for one subtree before the other:

```
Node::Then(left, right) => {
    // do left subtree, which stores result in r[stack_ptr]
    let mut instr = translate_subtree(left, stack_ptr, next_label);
    // do right subtree, which stores result in r[stack_ptr]
    instr.append(&mut translate_subtree(right, stack_ptr, next_label));

    instr
}
```

Conditional nodes, such as `If`, also make use of labels:

```
Node::If(cond, if_not_zero, if_zero) => {
    let label_else = *next_label;
    let label_after = label_else + 1;
    *next_label += 2;
    // process cond subtree, store result in r[stack_ptr]
    let mut instr = translate_subtree(cond, stack_ptr, next_label);

    // if cond != 0, keep going to the if_not_zero block, then jump to label_after
    instr.push(Instruction::Jump(label_else, JumpCondition::Zero(stack_ptr)));
    instr.append(&mut translate_subtree(if_not_zero, stack_ptr, next_label));
    instr.push(Instruction::Jump(label_after, JumpCondition::None));
    // else, jump to label_else, do the if_zero block, then continue to label_after
    instr.push(Instruction::Label(label_else));
    instr.append(&mut translate_subtree(if_zero, stack_ptr, next_label));
    instr.push(Instruction::Label(label_after));

    instr
}
```

Finally, nodes making use of special registers such as the `RESULT_REGISTER` and the `COMPARE_REGISTER` will move results to `r[stack_ptr]` to make them accessible by their parents.

```
Node::Turtle(operation) => {
    vec!(
        // do the operation, storing result in r[RESULT_REGISTER]
        Instruction::Turtle(*operation),
        // copy r[RESULT_REGISTER] into r[stack_ptr] to move it up the tree
        Instruction::Binary(stack_ptr, Source::Register(RESULT_REGISTER), Set),
    )
}
```

For example, the tree below translates into the instructions below it.

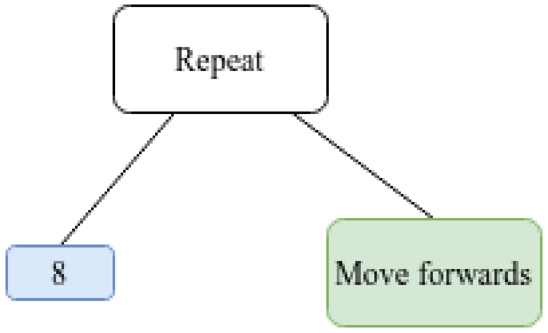

Figure 3.4: A simple tree.

```
Binary(101, Value(8), Set),  // set reg. 101 to 8
Label(0),
Jump(1, Zero(101)),  // jump to label 1 when reg. 101 is 0
Turtle(Move(Forward)),
Binary(100, Register(200), Set),  // set reg. 100 to the value of the result register
Unary(101, Decrement),  // decrement reg. 101
Jump(0, None),  // jump to label 0 always
Label(1)
```

Figure 3.5: The equivalent list of instructions.

3.4 Genetic algorithm

For a brief summary, a genetic algorithm is an algorithm taking inspiration from the real-life processes of natural selection and evolution. It involves taking an entire “population” of “individuals”, in this case tree-structured programs, and evaluating them. The next generation is created from the current one, with individuals that performed better having a higher chance of “reproducing”. To make helpful changes, individuals are mixed together and randomly modified individually. Here is a short glossary to help with the following section:

- *population* or *generation*: a group of individuals evaluated, modified and propagated in a single iteration of the genetic algorithm

- *individual*: a single tree, in a population
- *evaluation*: the process of checking the performance of every individual in a generation
- *fitness*: the weight allocated to each individual during the selection process
- *selection*: the process of building a new population from the old one, by copying individuals or crossing them together
- *crossover*: the building of one individual out of parts of two parents
- *mutation*: using randomness to modify parts of an individual in hope of creating useful new traits

The over-all training function, that goes through the generations gradually improving individuals, is detailed below. This function takes as parameter a `target` that the trained individuals are supposed to build.

```rust
pub fn train(target: &BlockSpace) -> definitions::Individual {
    // randomly-generate the first generation
    let mut generation = Generation::random();
    // evaluate the first generation
    generation.evaluate(target);
    let mut best_overall = generation.population[0].clone();
    for _gen in 0..GEN_COUNT {
        // find and store the best individual from each generation
        let best_individual = &generation.population[generation.best_index.unwrap()];
        let best_result = &best_individual.result.unwrap();
        // store this individual if it is the best over-all
        if best_result.score > best_overall.result.unwrap().score {
            best_overall = best_individual.clone();
        }
        // select individuals from the current generation to either take part in the next
        // one directly or as parents for new children
        let (kept_over, parent_pairs) = generation.select_weighted_by_fitness();
        // build a new generation from the selected individuals
        generation = Generation::from_old(&generation, &kept_over, &parent_pairs);
        // produce random, potentially useful mutations in the individuals
        generation.mutate();
        // evaluate the built generation, for the next iteration
        generation.evaluate(target);
    }

    // return the best individual found in any generation
    best_overall
}
```

In the following sub-sections, I will go into more detail concerning each part of this algorithm.

3.4.1 Creating the first generation

The first generation is constructed by randomly-generating individuals using two methods and several depths. This method, proposed by J.R. Koza [2], is called "ramped half-and-half".

The first method is called *full*, and involves creating a full tree down to a certain depth. Nodes below that depth must have a non-zero arity (i.e. not be leaves), and the entire maximum depth must be made up of leaves. The second method, *grow*, permits leaves further up in the tree with a certain probability.

```
pub fn generate(method: Method, max_depth: usize) -> Node {
    recurse(method, max_depth, 0)
}

fn recurse(method: Method, max_depth: usize, current_depth: usize) -> Node {
    let mut rng = rand::thread_rng();

    if current_depth == max_depth || (method == Method::Grow && rng.gen_bool(P_GROW_LEAF)) {
        // generate a leaf, excluding useless leaves such as the Null node
        random_useful_leaf()
    } else {
        // generate a function, continuing recursion with its children
        random_function(method, max_depth, current_depth)
    }
}
```

One useful nuance here is that, even though there exist many types of unary and binary operations, the `Unary` and `Binary` nodes have the same chance to be generated as control structures like `Repeat`. This should, in theory, make it so that programs have a higher chance of generating the comparatively complex structures required to construct parts of structures.

The first generation is constructed using equal numbers of individuals generated using either method, and by varying the `max_depth` parameter between two thresholds (hence the 'ramped' part of the name). This results in a nice variety of small and large, simple and complex trees, hopefully providing a relatively unbiased start.

3.4.2 Evaluation

Trees are evaluated against a target structure, measuring how close the built structure is to the target. This measurement is done using the *Sørensen–Dice coefficient*[4][1], henceforth referred to as the "Dice index". This index is calculated as follows:

$$DSC = \frac{2TP}{2TP + FP + FN}$$

$$\begin{aligned} \text{where} \quad TP &= \text{number of true positives} \\ FP &= \text{number of false positives} \\ FN &= \text{number of false negatives} \end{aligned}$$

To be more explicit, a "true positive" in this case means a position that is filled in both the target structure and the built one (and similar for the other two terms). This index has a value of 0 when the two have nothing in common, and 1 when they are identical (hence, FP and FN are 0).

One thing that must be taken into account, however, is that a smaller program (shallower tree) should score better than a bigger one (deeper tree) which does the same thing. For this purpose, the actual score of a tree is the following:

$$s = DSC * \frac{DS + MD - d}{DS + MD}$$

$$\begin{aligned} \text{where} \quad DSC &= \text{Dice index} \\ DS &= \text{depth softener} \\ MD &= \text{the maximum allowed depth of a tree} \\ d &= \text{the depth of the tree} \end{aligned}$$

The *depth softener* is a parameter of the algorithm used to control the importance of tree's depth to the score of an individual; a bigger DS means that depth has less of an impact.

Individuals whose tree exceeds the allowed depth, or whose program either takes too long or terminates with an error, receive a score of $-\infty$.

The `result` field of each individual is made up of its `score`, its unmodified `dice_index`, and a boolean value `perfect` which is set to true if it has a Dice index of 1.

After evaluating each individual, a generation will also store the best score of any individual in the population, as well as the worst finite score found.

3.4.3 Selection

Selection is the process of picking individuals from the current generation to use in the creation of the next one. These individuals are chosen based on their score in one of two ways.

Tournament selection involves running a number "tournaments" and picking the winner as a parent for the next generation. A tournament involves randomly selecting a handful of individuals from the population, and sorting them by their results. Then, the best individual in this set is chosen with probability p, the second with probability $(1-p)p$, etc. If no other individual is chosen, the worst one will be.

Tournament selection only picks parents, so the entirety of the next population will be constructed using crossover.

Fitness-weighted selection, on the other hand, is done on the whole population at once. First, each individual's score s is mapped into a fitness f using the following formula:

$$f = \left(\frac{s - s_{min}}{s_{max} - s_{min}} \right)^{SP}$$

$$\begin{aligned} \text{where} \quad s_{min} &= \text{the worst score in the generation} \\ s_{max} &= \text{the best}^{1}\text{score in the generation} \\ SP &= \text{selection pressure} \end{aligned}$$

[1]If every individual in the generation has the same score, then s_{max} will be equal to $s_{min} + 0.1$ instead, in order to avoid division by zero.

The selection pressure is a parameter that dictates how strongly the selection process is weighed in favor of better individuals.

After the fitness of each individual is calculated, the function uses them as weights to randomly pick (with replacement) a number of individuals from the population to either be copied directly to the next generation, or to serve as parents during the crossover process.

3.4.4 Crossover

To easily explain crossover, both in my code and this sub-section, I will borrow a couple of terms from the field of the horticultural technique of *grafting*, which involves joining tissue from two different plants in order to continue their growth together. In this context, the lower part of the combined plant is called a rootstock or *stock*, while the upper part is called a *scion*. I, however, will be using the two terms to refer to the two trees that will provide the necessary parts, the parents of the new child.

Crossover, for the purposes of genetic algorithms, involves taking two individuals and mixing parts of them together to create a new individual. This is analogous to the real-life genetic process of chromosomal crossover, where matching regions on matching chromosomes break apart and reconnect with each other.

In my code, crossover involves the following process:

- pick a weighted-random node of the stock, with leaves having a weight of 1 and other nodes a weight of 9, a method again suggested by Koza[2]
- randomly descend the scion to a depth equal to that of the selected node (stopping if a leaf node is hit)
- choose a random node from the resulting subtree of the scion, weighing the crossover process in favor of exchanging less genetic material
- replace the selected stock node with the chosen node from the scion

```
fn crossover(stock: &Node, scion: &Node) -> Node {
    let mut stock = stock.clone();
    let (stock_point, stock_point_depth) = stock.get_weighted_node_mut();

    // randomly descend as many levels as the stock point is deep, *then* choose a random node
    let scion_point = scion
```

```
        .randomly_descend(stock_point_depth)
        .get_weighted_node();

    let mut new_branch = scion_point.clone();
    std::mem::swap(stock_point, &mut new_branch);

    stock
}
```

This operation effectively replaces a subtree of the stock with that one of the scion's.

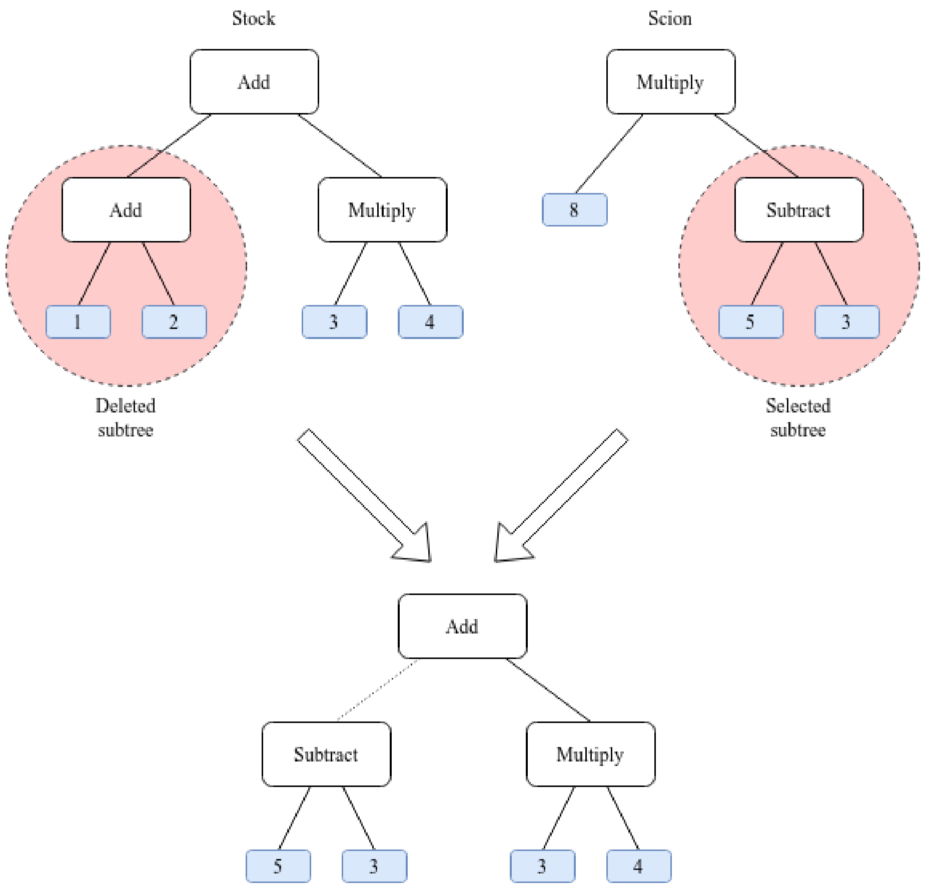

Figure 3.6: A stock and a scion combine to produce a single child.

3.4.5 Mutation

Mutation is a process that can modify a tree unpredictably, with the idea of producing useful changes. My implementation applies two different kinds of mutation for each individual, described as follows.

One kind of mutation applies to individual nodes. For each node in an individual, there is a certain probability to be replaced by another node with the same arity:

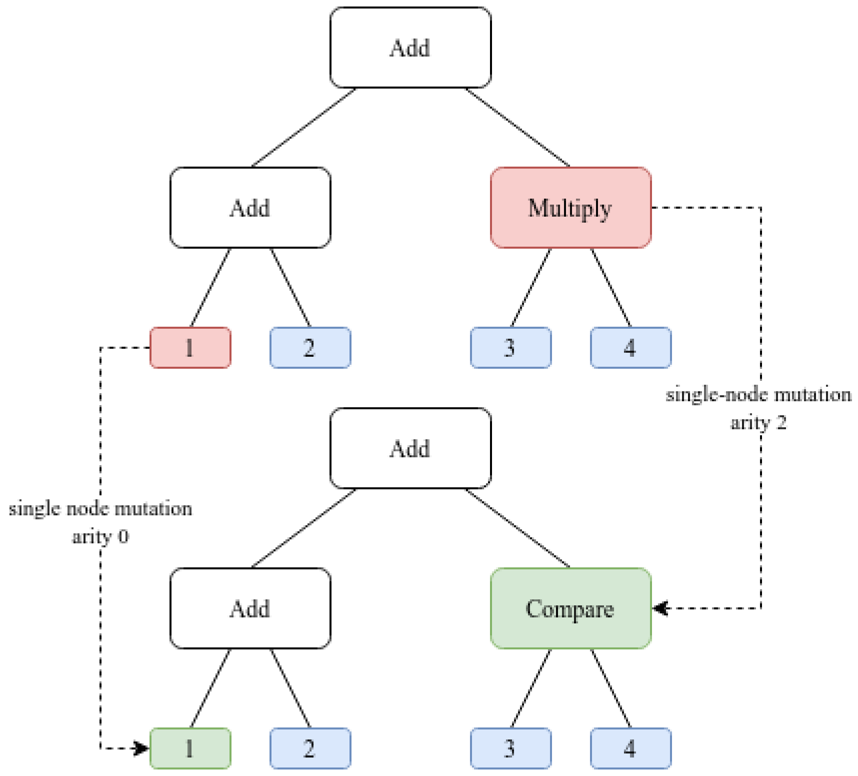

Figure 3.7: The red nodes are replaced by the green ones due to single-node mutation. The rest of the nodes remain unchanged.

The other kind of mutation involves replacing entire subtrees with newly generated ones. This happens once per tree, with a certain probability. Specifically, this involves picking a weighted-random node (similarly to subsection 3.4.4). This node is the root of a subtree which will be replaced by a new subtree, generated using the *grow* method up to the same depth as the old one.

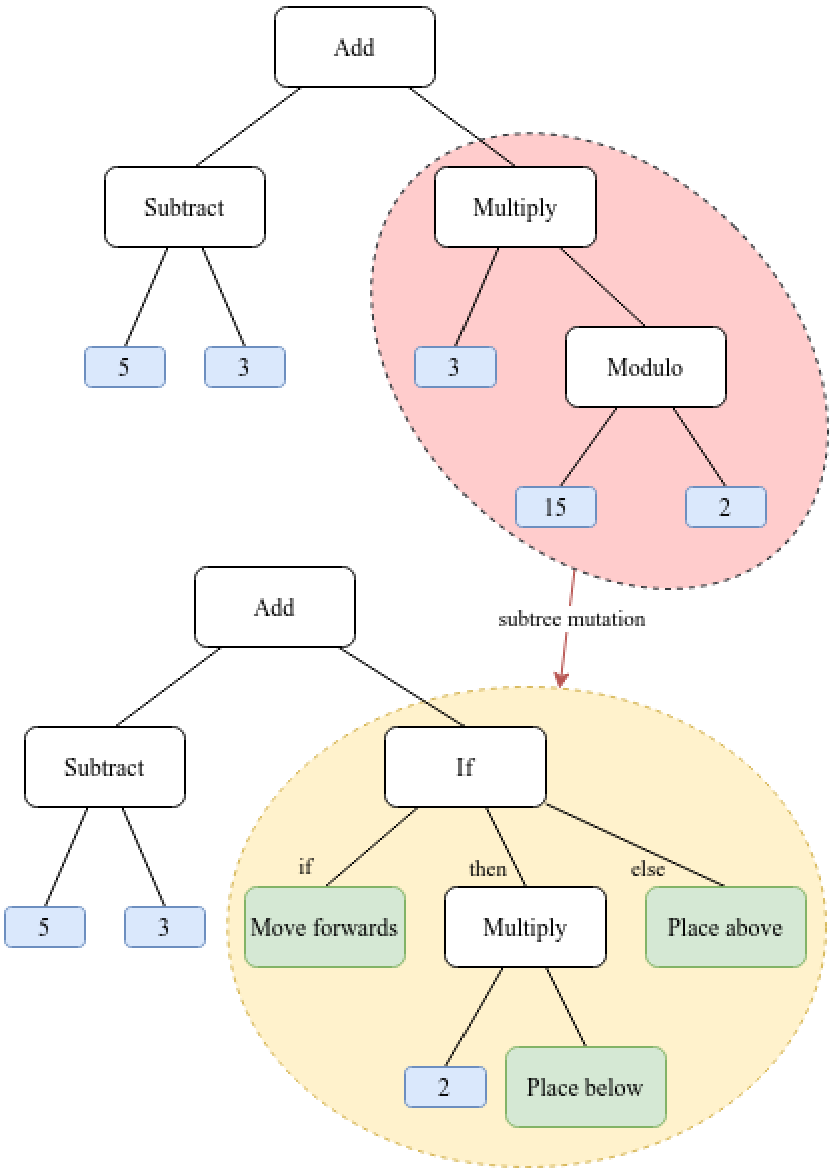

Figure 3.8: The red subtree is replaced by the yellow one due to subtree mutation. The rest of the nodes remain unchanged.

Chapter 4

Experiments

Chapter 5

Results and conclusions

Bibliography

[1] L. R. Dice. Measures of the amount of ecologic association between species. *Ecology*, 26(3):297–302, 1945.

[2] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.

[3] ComputerCraft community. *ComputerCraft Wiki*, Sept. 2012. Accessed: 14 jun 2021.

[4] T. J. Sørensen. *A method of establishing groups of equal amplitude in plant sociology based on similarity of species content and its application to analyses of the vegetation on Danish commons*, volume 5. Munksgaard Copenhagen, 1948.