Genetic programming for building voxel structures with turtle robots

Iacobescu Tudor

2021

Contents

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[4] 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*[6], 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[5] 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.

Over the course of this thesis, I will be exploring exactly what my training and testing environment consists of. I will be using this environment to run several tests, and analyzing their results. I will also explain why I think the performance of the algorithm is unsatisfactory, and why it isn't too useful in practice.

Contributions

The idea behind the thesis was constructed collaboratively with my guiding professor, Assist. dr. Croitoru Eugen. The Rust source code for the training environment and the Python code for the two helper scripts were written by

me. Screenshots are taken by me in Minecraft[4] using CC: Tweaked[1], a fork of ComputerCraft[6]. Diagrams are made by me using draw.io. Test running, result collection and analysis are all my own work as well.

Chapter 1

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.

1.1 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[7], 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.

1.2   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.

For clarity, and for fun, I'm going to use screenshots of Minecraft to illustrate the training space.
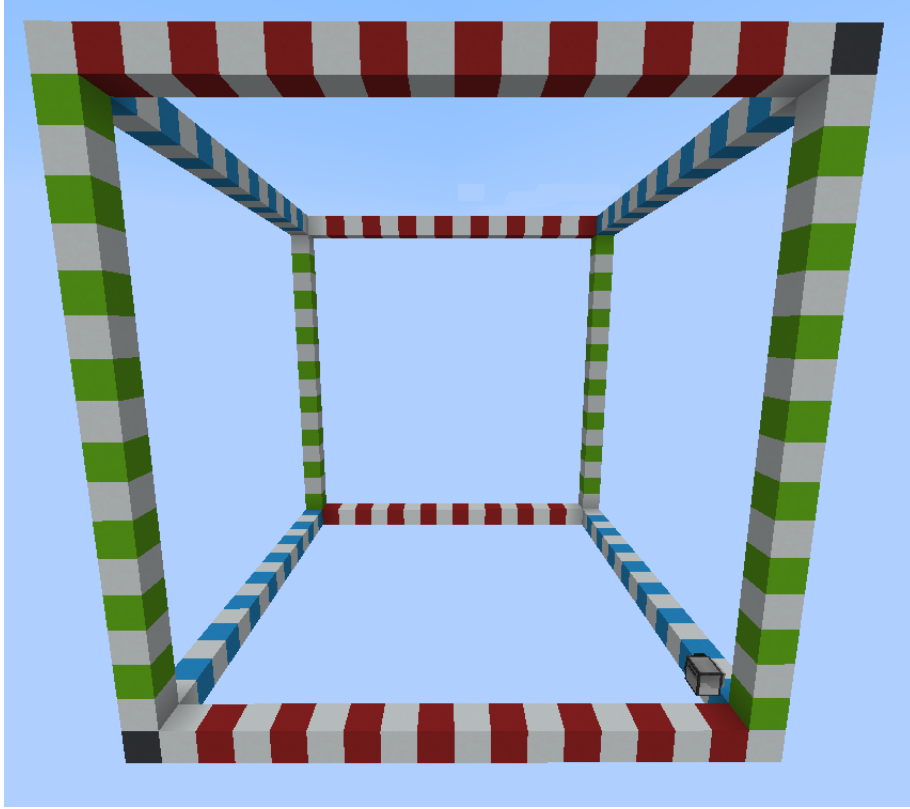
Figure 1.1: The training space, as it would appear in Minecraft. The axes are represented by the red (x), green (y) and blue (z) edges of the cube, and the training space is contained within. The turtle is located at $(0, 0, 0)$, and is facing towards positive x.

Chapter 2

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.

2.1 The simulator

The simulator for the training space and the turtle itself is fairly simple.

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

Listing 2.1: The `Simulator` struct.

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];
```

Listing 2.2: The `BlockSpace` type alias.

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
}
```

Listing 2.3: The `Turtle` struct.

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, Listing 2.4 contains 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
    }
}
```

Listing 2.4: The code for attempting to move the turtle.

2.2   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 structure in Listing 2.5.

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

```
    simulator: Simulator,
}
```

Listing 2.5: The `Program` struct.

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, as seen in Listing 2.6.

```
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
}
```

Listing 2.6: The `Instruction` enum.

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` (Listing 2.7), on the other hand, holds operations specific to the problem domain, as described in section 1.1.

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

Listing 2.7: The `TurtleOperation` enum.

To execute the code, the VM goes through instructions step by step, processing each one. For example, the code snippet in Listing 2.8 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);
}
```

Listing 2.8: The handler for the `Unary` instruction.

One can notice 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 (see Listing 2.9).

```
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,
    });
}
```

Listing 2.9: The handler for the `Compare` instruction.

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 (Listing 2.10).

```
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);
```

13

```
        if destination == None {
            return Err("Attempted to jump to missing label");
        }
        let destination = destination.unwrap();
        self.instruction_pointer = *destination;
        jumped = true;
    }
},
```

Listing 2.10: The handler for the `Jump` instruction.

Lastly, turtle operations interact with the simulator and place their results in the `RESULT_REGISTER` (Listing 2.11).

```
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);
        },
    }
}
```

Listing 2.11: The handler for the `Turtle` instruction.

2.3 Tree-based programs

2.3.1 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.

SET REGISTER	[Target register]	[Source register]	

mutation

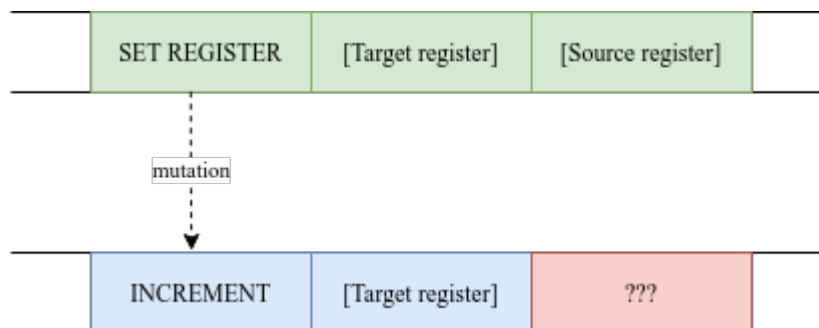| INCREMENT	[Target register]	???	

Figure 2.1: 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.

2.3.2 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 listed in Listing 2.12.

```rust
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),
}
```

Listing 2.12: The `Node` enum.

I will also be representing these trees graphically, to aid in their understanding for the purposes of this thesis. For example, the literal in Listing 2.13 is equivalent to the tree pictured in Figure 2.2:

```rust
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))),
```

```
    )),
)
```

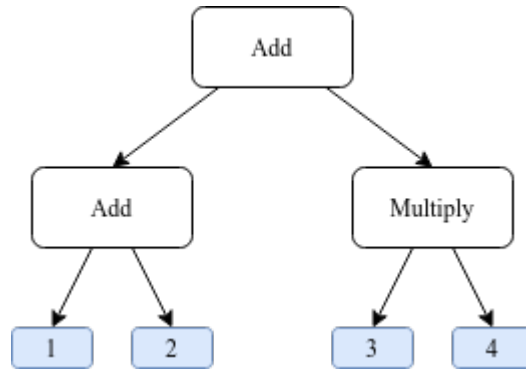Listing 2.13: Tree literal equivalent to Figure 2.2

Figure 2.2: 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.
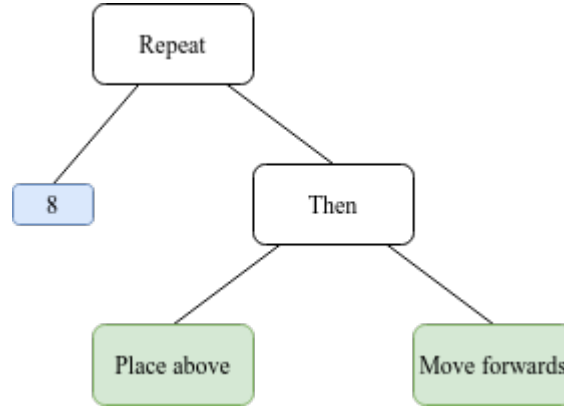

Figure 2.3: A simple program that builds a line from $(0, 1, 0)$ to $(7, 1, 0)$.

2.3.3 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 uses the functions shown in Listing 2.14, where the second function `translate_subtree` is a recursive function.

```rust
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 {
        ...
    }
}
```

Listing 2.14: The `translate_tree` and `translate_subtree` functions.

For example, in Listing 2.15, 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.

```rust
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
}
```

Listing 2.15: `Unary` node translation.

For multiple subtrees, the stack pointer is incremented, so that two separate values may be used, as in Listing 2.16.

```rust
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
}
```

Listing 2.16: `Binary` node translation.

A `Then` node simply adds the instructions for one subtree before the other, like in Listing 2.17

```
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
}
```

Listing 2.17: `Then` node translation.

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
}
```

Listing 2.18: `If` node translation.

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),
    )
}
```

Listing 2.19: `Turtle` node translation.

For example, the tree in Figure 2.4 translates into the instructions in Listing 2.20.
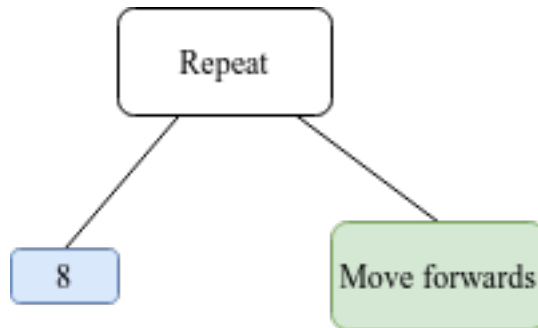

Figure 2.4: A simple tree.

```
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)
```

Listing 2.20: The equivalent list of instructions.

2.4 Genetic algorithm

For a brief summary, a genetic algorithm[3] 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

- *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 in Listing 2.21. This function takes as parameter a `target` that the trained individuals are supposed to build.

```rust
pub fn train(target: &BlockSpace) -> definitions::Individual {
    // generate initial generation
    let mut generation = Generation::random();
    generation.evaluate(target);
    let mut best_overall = generation.population[0].clone();
    for _gen in 0..GEN_COUNT {
        // store best individual of generation
        let best_individual =
            &generation.population[generation.best_index.unwrap()];
        let best_result = &best_individual.result.unwrap();
        // replace best_overall if better individual found
        if best_result.score > best_overall.result.unwrap().score {
```

```rust
                best_overall = best_individual.clone();
            }
            // use either selection method to pick out individuals for the next gen
            let (kept_over, parent_pairs) = match SELECTION_METHOD {
                SelectionMethod::FitnessWeighted =>
                    generation.select_weighted_by_fitness(),
                SelectionMethod::Tournament =>
                    generation.select_by_tournament(),
            };
            // create the new generation (through cloning and crossover)
            generation =
                Generation::from_old(&generation, &kept_over, &parent_pairs);
            // cause potentially useful mutations
            generation.mutate();
            // evaluate new generation
            generation.evaluate(target);
        }


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

Listing 2.21: The training function.

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

2.4.1   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.

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

```rust
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)
    }
}
```

Listing 2.22: The generation function, which can use either method.

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.

2.4.2 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*[9][8], henceforth referred to as the "Dice index". This index is calculated as follows:

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

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

24

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:

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

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

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.

2.4.3   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.

Fitness proportionate selection (referred to in code as "fitness-weighted"), 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:

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

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

The selection pressure is a parameter that dictates how strongly the selection process is weighed in favor of better individuals. The fitness value is calculated this way to normalize the chances for every individual to be chosen, giving even the worst individuals some chance to be picked.

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.

2.4.4 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.

---

[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.

In my code (Listing 2.23), 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

```rust
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
}
```

Listing 2.23: The code handling the crossover.

This operation effectively replaces a subtree of the stock with that one of the scion's.
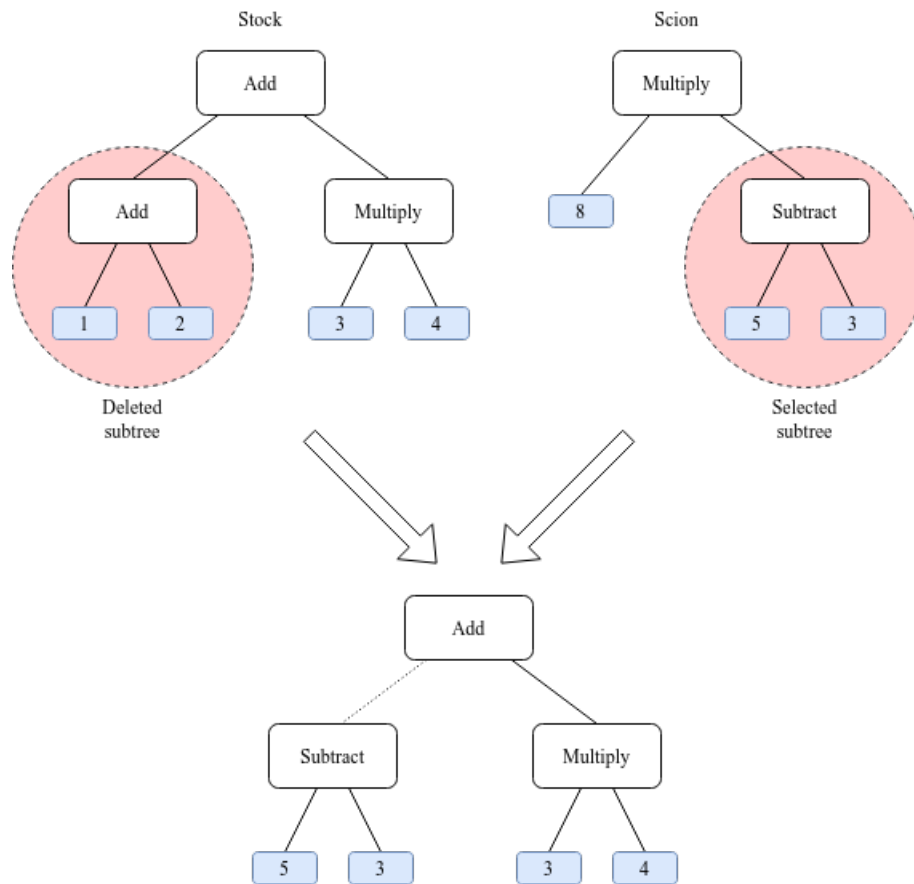
Figure 2.5: A stock and a scion combine to produce a single child.

2.4.5 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:
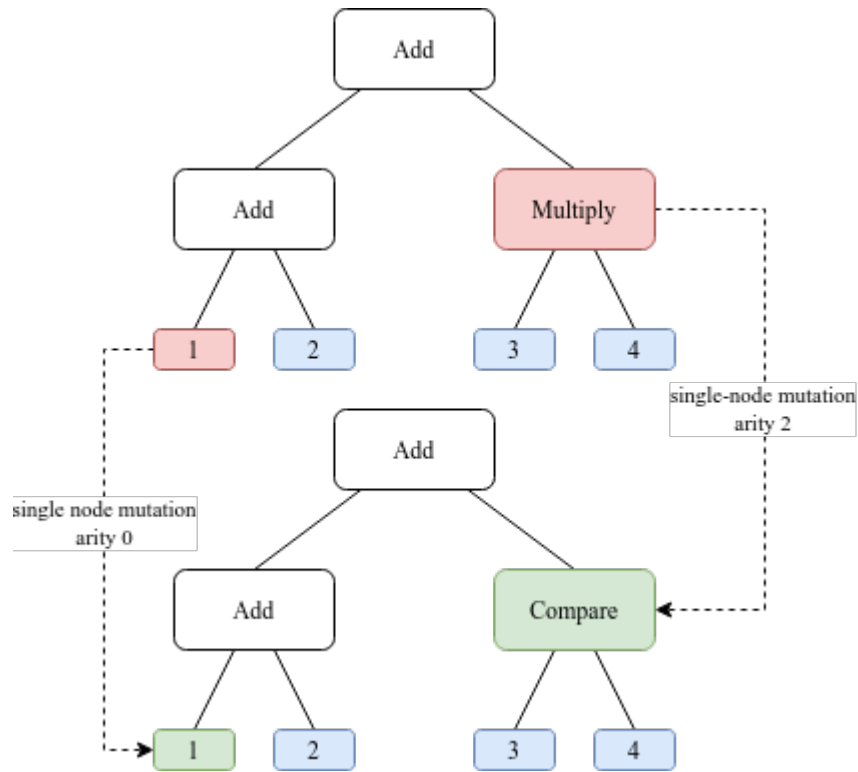
Figure 2.6: The red nodes are replaced by the green ones due to single-node mutation. The rest of the nodes remain unchanged.

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 2.4.4). 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.
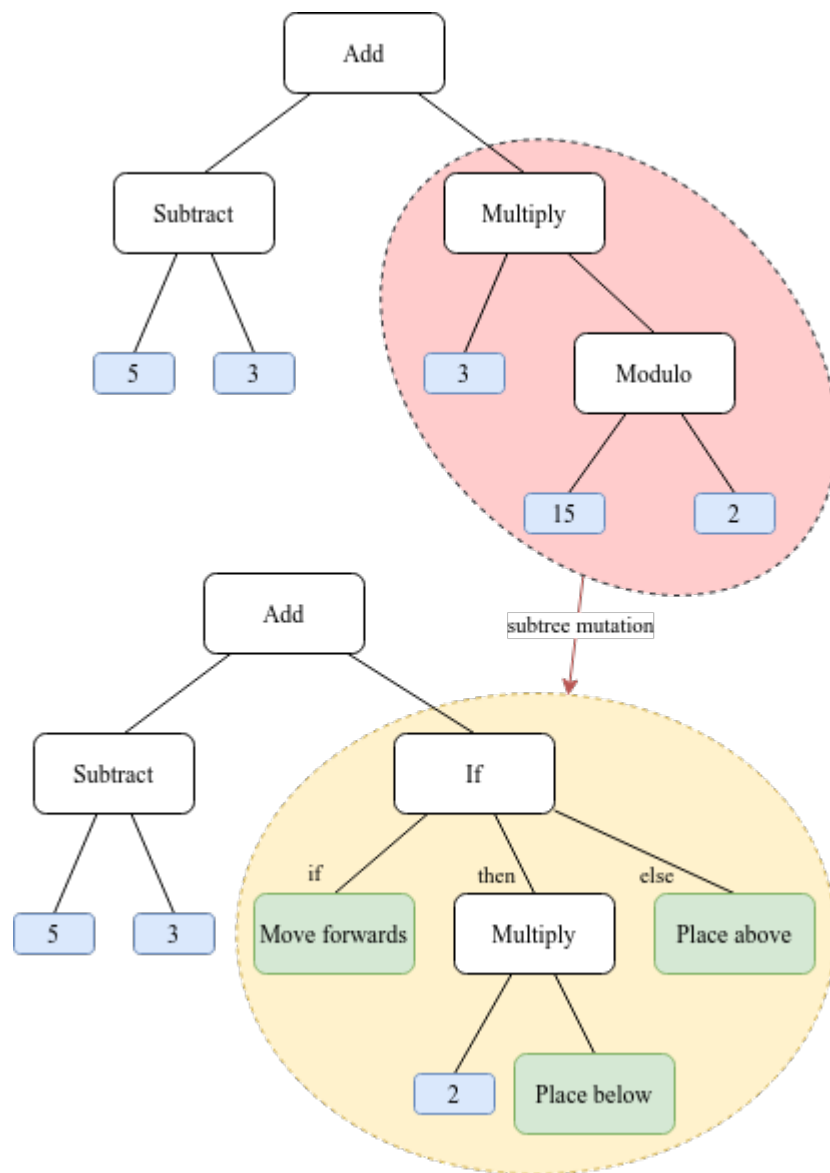
Figure 2.7: The red subtree is replaced by the yellow one due to subtree mutation. The rest of the nodes remain unchanged.

Chapter 3

Experiments

In this chapter, I will be exploring the experiments I have conducted using the program described thus far. I will be illustrating them, analyzing their results individually, and then comparing them to each other.

3.1   Experimental setup

To obtain rigorous results, I am using a "meta-search" script, written in Python. This script will iterate through all possible combinations of the parameters I seek to explore, running 30 training runs with each one. Then, I will be averaging out the 30 runs to produce two values: the average Dice index of the best individual produced by each run, as well as the average depth of said individual's tree.

Each run will use a number of fixed parameters, including the following:

- a population of 50 individuals (5 individuals per generation method and per depth between 2 and 6, as per the ramped half-and-half technique)

- a run-time of 1000 generations

- a maximum run-time of 10 milliseconds for each evaluated program

- a maximum depth of 12 for each individual's tree

For each run, the following parameters will vary:

- the "depth softener" parameter (described in subsection 2.4.2), with values in $\{1.0, 6.0, 10.0\}$

- the chance for a subtree mutation, with values in $\{0.0, 0.001, 0.01, 0.02, 0.05\}$

- the chance for a single-node mutation (which can happen multiple times per tree), with values in $\{0.0, 0.001, 0.01, 0.05, 0.1\}$

- the proportion of each new population that is generated through crossover instead of cloning, with values in $\{0.5, 0.75, 0.9\}$

- the method used for selection, which can be either `Tournament` or `FitnessWeighted`

Additionally, each selection method has certain parameters which will also vary:

- tournament selection has a tournament size parameter (in $\{2, 4, 7\}$) and a probability p described in subsection 2.4.3 (in $\{0.8, 1.0\}$)

- fitness proportionate selection has a selection pressure parameter with values in $\{0.2, 1.0, 5.0\}$

Given all parameters that can vary, the total number of combinations that must be tried is 675 for fitness proportionate selection, and 1350 for tournament selection.

3.2   Test cases

3.2.1   Elementary case: a single block

This test case is the simplest, with the target structure made up of a single block immediately above the starting position of the turtle. It can easily be solved optimally by a single-node tree consisting of just a `Place(UP)` node.
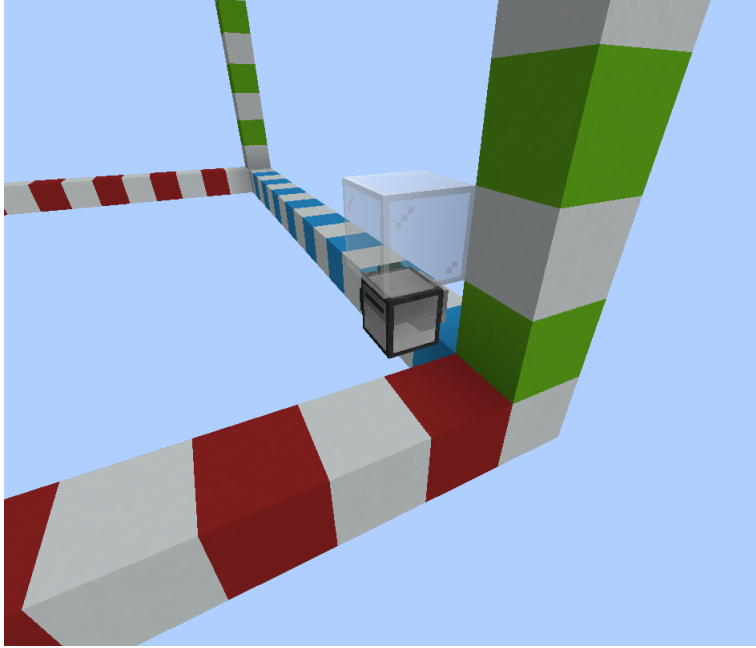
Figure 3.1: A screenshot showing the elementary case. The white glass block represents the target structure.

Both selection methods easily obtained the expected result, a Dice index of 1.0 and an average depth of 1.0, with most of the parameter combinations.

3.2.2 Second case: two blocks

The second example is slightly more complicated, with a second block added in front of the first one. This means that even the simplest solution would require at least three instructions, and even more nodes.
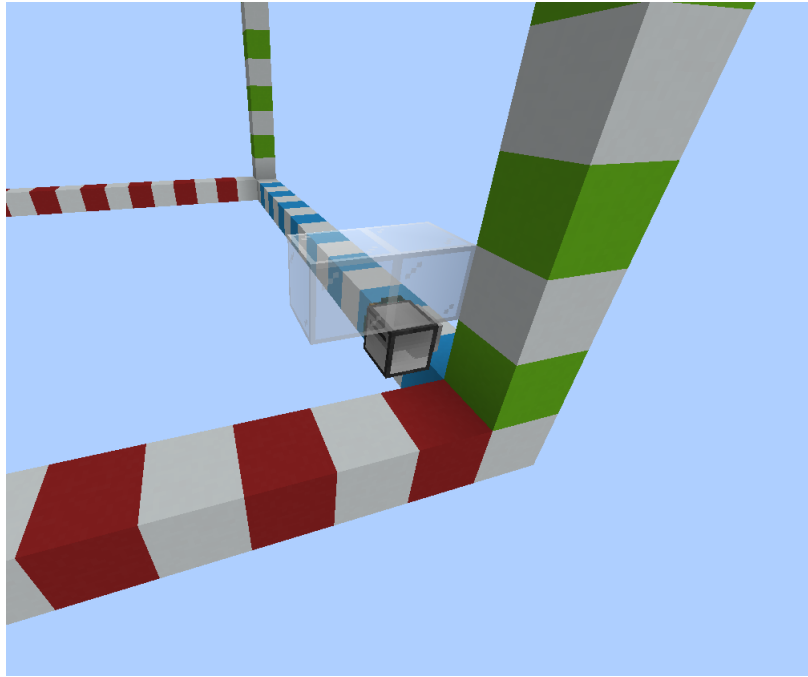
Figure 3.2: A screenshot showing the second case. There is another block to be placed in front of the first one.

The effect of the added complexity is visible immediately: the best fitness proportionate selection combination gets an average Dice index of 0.855, while the best tournament one gets an even weaker result, of 0.8.

3.2.3   Third case: four blocks

The third example adds two more blocks, creating a short line of four. This case is where one might reasonably expect loop structures to become useful. However, the results are not any better than before.
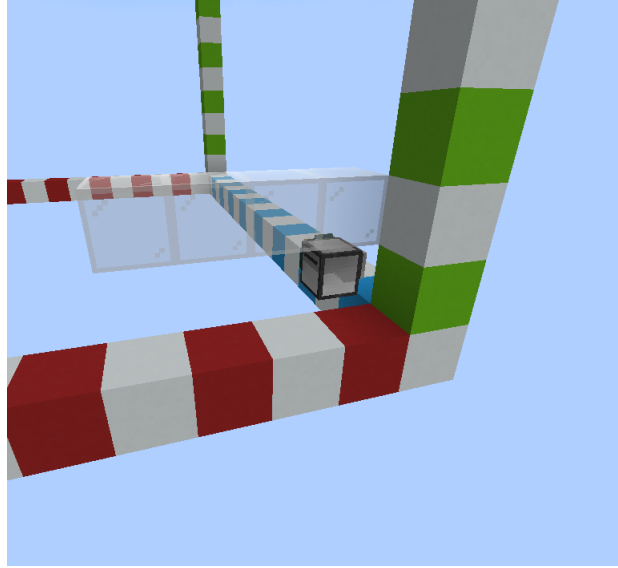
Figure 3.3: A screenshot showing the third case. Two more blocks are present compared to the previous case.

The best fitness proportionate selection combination obtains an average index of 0.616, and the best tournament one, a weaker result of 0.582.

Given that test case 4 has its results greatly affected by a local maxima, and test case 5 performs very poorly regardless of parameters (both detailed below), case 3 is probably the best fit for analyzing the impact of each parameter on the performance of the algorithm. For this purpose, the output files for each case and selection method contain not just the best combination of parameters, but all possible combinations and their results. With a second python script, these results may be analyzed; the outputs of this script, edited for readability, are Listing 3.1 and Listing 3.2.

```
depth softener
     1.0: 0.4458, 1.4754
     6.0: 0.4713, 1.7945
    10.0: 0.4812, 1.9041
subtree mutation chance
     0.0: 0.4560, 1.7173
   0.001: 0.4536, 1.6773
    0.01: 0.4580, 1.5988
    0.02: 0.4669, 1.6835
    0.05: 0.4960, 1.9467
```

```
single-node mutation chance
     0.0: 0.4600, 1.7096
   0.001: 0.4646, 1.7560
    0.01: 0.4703, 1.6864
    0.05: 0.4713, 1.7526
     0.1: 0.4644, 1.7188
crossover size proportion
     0.5: 0.4715, 1.7661
    0.75: 0.4656, 1.7145
     0.9: 0.4612, 1.6935
selection pressure
     0.2: 0.4340, 1.5099
     1.0: 0.4668, 1.8507
     5.0: 0.4975, 1.8135
```

Listing 3.1: Average Dice index and depth for each parameter of the fitness proportionate selection runs.

```
depth softener
     1.0: 0.4403, 1.3321
     6.0: 0.4539, 1.4961
    10.0: 0.4565, 1.5313
subtree mutation chance
     0.0: 0.4466, 1.4711
   0.001: 0.4496, 1.4746
    0.01: 0.4514, 1.4401
    0.02: 0.4498, 1.4188
    0.05: 0.4536, 1.4614
single-node mutation chance
     0.0: 0.4471, 1.4414
   0.001: 0.4482, 1.4274
    0.01: 0.4500, 1.4188
    0.05: 0.4573, 1.4970
     0.1: 0.4485, 1.4814
crossover size proportion
     0.5: 0.4491, 1.4307
    0.75: 0.4498, 1.4461
     0.9: 0.4519, 1.4827
tournament size
       2: 0.4437, 1.5727
```

```
      4: 0.4541, 1.4336
      7: 0.4529, 1.3533
tournament p
    0.8: 0.4483, 1.4585
    1.0: 0.4521, 1.4479
```

Listing 3.2: Average Dice index and depth for each parameter of the tournament selection runs.

This analysis reveals that while differences in the performance of each parameter do exist, they are not very large. The parameters have a bigger impact in the fitness proportionate runs, but even there the biggest difference recorded is between `selection_pressure` values 0.2 (with an average index of 0.4340) and 5.0 (0.4975).

3.2.4 Fourth case: a line of eight blocks

Four more blocks are added, expanding the line to eight blocks. This means that looping is now almost mandatory for any decent results.
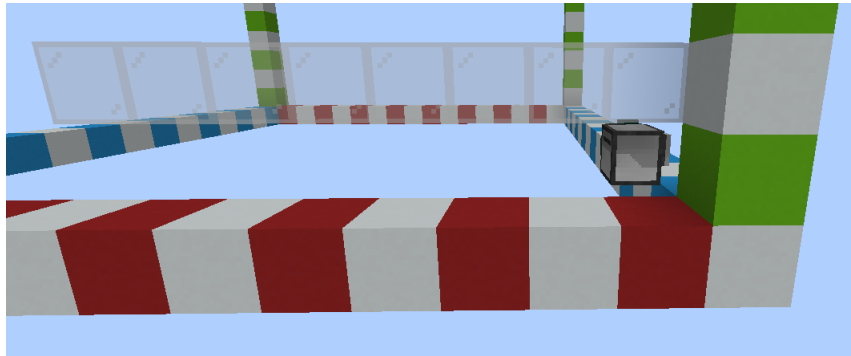

Figure 3.4: A screenshot of the fourth case, with eight blocks in total.

The results are, in fact, better than the case before. The best fitness proportionate combination yields an average index of 0.666 with an average depth of 2.000. The best tournament one is slightly weaker, with an average Dice index of 0.645 and a depth of 2.366. To analyze the reason for the better results, I ran 30 more tries with each combination.

The first combination, using fitness proportionate selection, was Listing 3.3.

```
{
  'crossover_size_proportion': 0.9,
```

```
 'depth_softener': 10.0,
 'mutation_chance': 0.05,
 'mutation_single_chance': 0.1,
 'selection_method': 'FitnessWeighted',
 'selection_pressure': 1.0,
}
```

Listing 3.3: A set of parameters that lead to an average index of 0.666 and an average depth of 2.000.

Running the program with this combination produces, overwhelmingly, the tree pictured in Figure 3.5.
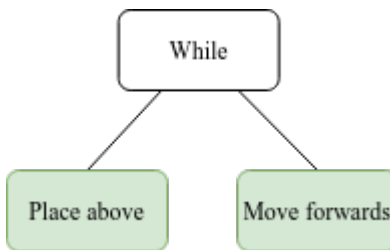

Figure 3.5: The tree which produces the expected results.

This tree is very simple, and what it does in practice is fill in an entire line, from $x = 0$ to $x = 15$, as pictured in Figure 3.6. It gets a Dice index of 2/3, because it produces 8 true positives and 8 false positives. This relatively high index, combined with the tree's great simplicity, makes it a local maxima that the algorithm falls into almost every time.
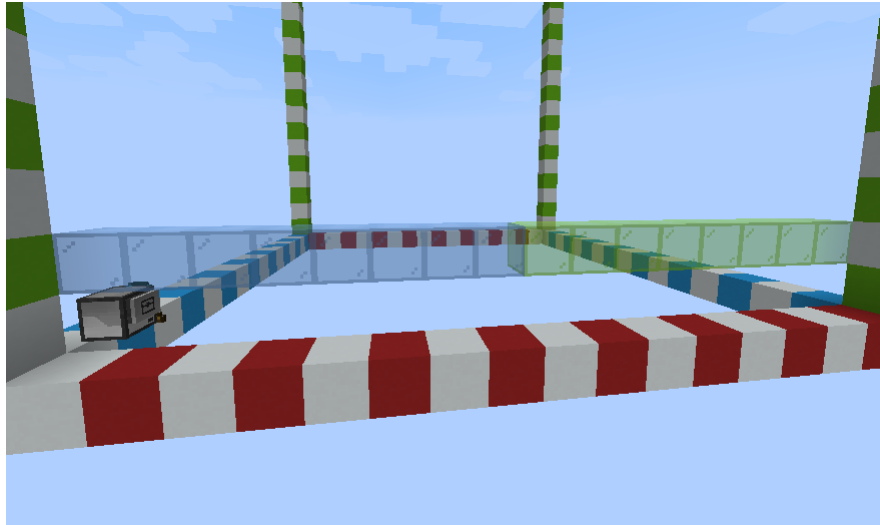
Figure 3.6: The structure constructed by the turtle. The true positives are represented by green glass, and the false positives, by blue glass. The turtle has stopped at the end of the testing space, unable to advance further because of the space's walls. This meant that there was a block above it on the next iteration of the loop, which caused the "Place above" call to fail and the program to stop.

The tournament combination, whose parameters are listed in Listing 3.4, was less consistent than its fitness proportionate counterpart, but the trees it produced that had the highest Dice index were the same ones, pictured in Figure 3.5.

```
{
  'crossover_size_proportion': 0.5,
  'depth_softener': 10.0,
  'mutation_chance': 0.05,
  'mutation_single_chance': 0.1,
  'selection_method': 'Tournament',
  'tournament_p': 0.8,
  'tournament_size': 2
}
```

Listing 3.4: A set of parameters that lead to an average index of 0.645 and an average depth of 2.366.

3.2.5 The hardest case: a cuboid

As revealed thus far, the algorithm does not fare very well even on simple test cases, with contrivances in its favor. The next test case, pictured in Figure 3.7, is closer to what this algorithm might actually be applied to: an arbitrary cuboid, further away in space.
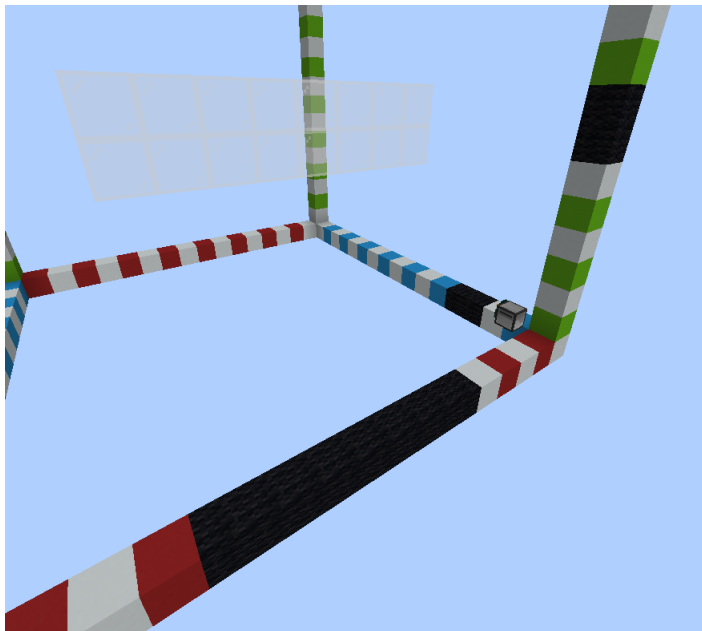

Figure 3.7: A screenshot of the fifth case, with a 7 by 2 by 2 cuboid from $(4, 6, 2)$ to $(11, 7, 3)$. The projections of this cuboid on the axes are highlighted as black segments, to clarify its size and position.

The results of this case are significantly weaker, with a Dice index of 0.005 (and an average depth of 2.833) for the fitness proportionate combination, and 0.002 (with a depth of 2.800) for the tournament one. This means that, even in the best cases, the algorithm only produced simple programs whose built structures just barely intersect the targets.
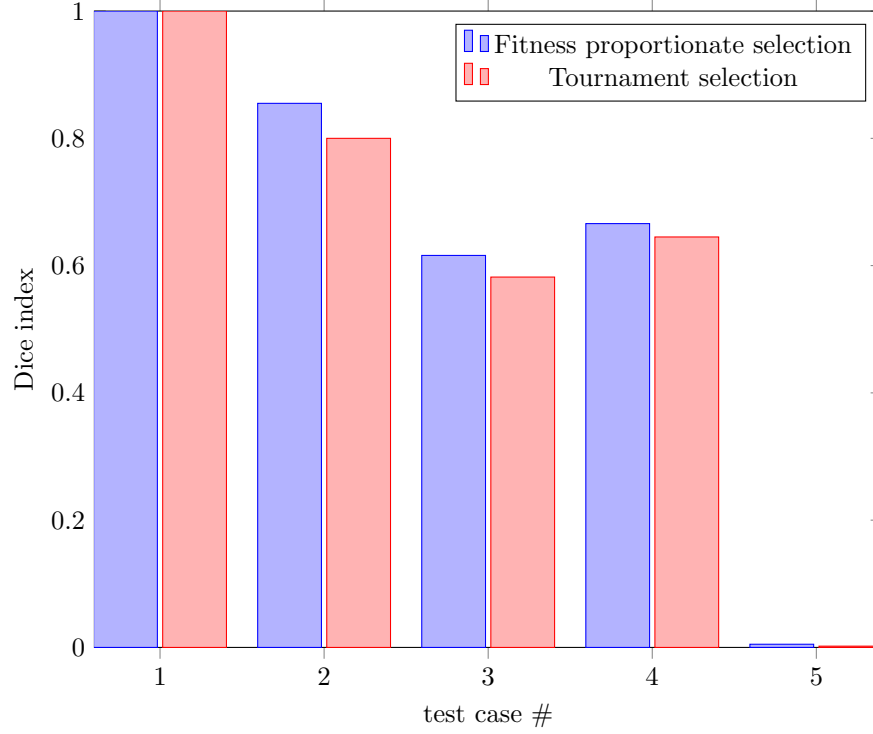
3.3 Overall analysis

Figure 3.8: A graph showing the average Dice index for the best parameter combination in each case, for each selection method.

Figure 3.8 follows the performance of each selection method for all five test cases. One thing that is apparent is that tournament selection is consistently worse-performing than fitness proportionate selection.

Apart from this, case 5 also makes it immediately clear that the algorithm performs very poorly in realistic applications. Even with the more idealized cases 1 through 4, the algorithm could not perfectly replicate the target structure in any cases except for the simplest one.

Lastly, one interesting thing to note, as previously mentioned, is that the algorithm performs better in test case 4 than it does in case 3. This is likely because of the aforementioned easy-to-reach local maxima (Figure 3.5), which takes advantage of a loop to easily yield comparatively good results.

Conclusions

Possible improvements and alternatives

The results of the tests are pretty clear: my implementation yields unsatisfactory results for basic test cases, and does not even come close to replicating the target structure for more complex ones. While I cannot exclude programming errors in the code I have written as causes for this poor performance, I can give some other possible causes.

It appears likely that the search space is too large: 256 registers and a large, flexible set of instructions lead to a very large number of possible trees that can be reached by the genetic algorithm. If this is a factor, it's possible that lowering the number of registers and shrinking the set of possible instructions would help; perhaps specializing the approach for only generating a certain kind of structure would lead to better results.

I believe it is also impractical to use genetic programming for simple structures in general. Lines, cuboids, and repeated patterns are simple applications of loops that are easy to write by hand. Circles, curves or ellipsoids would likely require mathematical functions that a genetic algorithm may have a hard time approximating evolutionarily.

A more complex structure, perhaps even using multiple kinds of voxels, would also be a bad match for this algorithm. A complex structure would be difficult to input into the algorithm, probably more so than constructing it by hand. More voxels would also greatly increase the search space, making it impractical to find the right solution.

If a complex structure needs to be replicated multiple times, it may be practical to instead build it once by hand, and use a "scanner" program to destroy it layer by layer, storing the location of each block in the structure into a 3D array, and then another "printer" program to re-build it in a way similar to

the operation of a 3D printer. However, a successful implementation of a genetic programming algorithm may yield improvements in the runtime of the "printer" program by generating a more efficient route than one iterating through every position.

Personal notes

Even though this project may not have yielded many useful results, I found it very interesting to work on. Firstly, it provided me an opportunity to expand on my knowledge of genetic algorithms from the courses and seminars I'd done previously. Even though this particular challenge may not be a very good fit for this kind of algorithm, there are still many other applications where it often beats other search strategies, such as function optimization or fields where the nature of the problem and the search space isn't entirely understood.

On the other hand, it also gave me a great opportunity to work with Rust, a language which I'd been meaning to explore for a while, but couldn't due to time constraints and college work. I'm thankful for the chance to try and mix my passion for programming with my favorite video game.

Bibliography

[1] Jonathan Coates. CC: Tweaked. URL: `https://tweaked.cc/`.

[2] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992. URL: `http://mitpress.mit.edu/books/genetic-programming`.

[3] Melanie Mitchell. *An introduction to genetic algorithms*. MIT press, 1998.

[4] Mojang. Minecraft, 2011. URL: `https://www.minecraft.net`.

[5] Riccardo Poli, William B Langdon, Nicholas F McPhee, and John R Koza. *A field guide to genetic programming*. Lulu. com, 2008.

[6] Daniel Ratcliffe. ComputerCraft. URL: `http://www.computercraft.info/`.

[7] ComputerCraft community. *Turtle (API) — ComputerCraft Wiki*, September 2012. Accessed: 14 jun 2021. URL: `http://www.computercraft.info/wiki/Turtle_(API)`.

[8] Lee Raymond Rice. Measures of the amount of ecologic association between species. *Ecology*, 26(3):297–302, 1945.

[9] Thorvald Julius 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.