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*, 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:

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

The blocks are stored in a 3D array.

```
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).

```
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),
}
```