Sudoku Solver Technical Specifications

Gerrit Roessler

2023

1 Introduction

This project exists to take an input file containing an (unsolved) sudoku board, and solve it. This will be a command line application developed mainly in the Rust language.

2 Goals

1. Command line input
2. Takes input file, generates an output file
3. Making use of the wave function collapse algorithm.

2.1 Stretch Goals

1. Graphical User Interface, or GUI
2. In the case of branching solutions, generating multiple outputs
3. Threading
4. GPU Acceleration

3 Wave Function Collapse

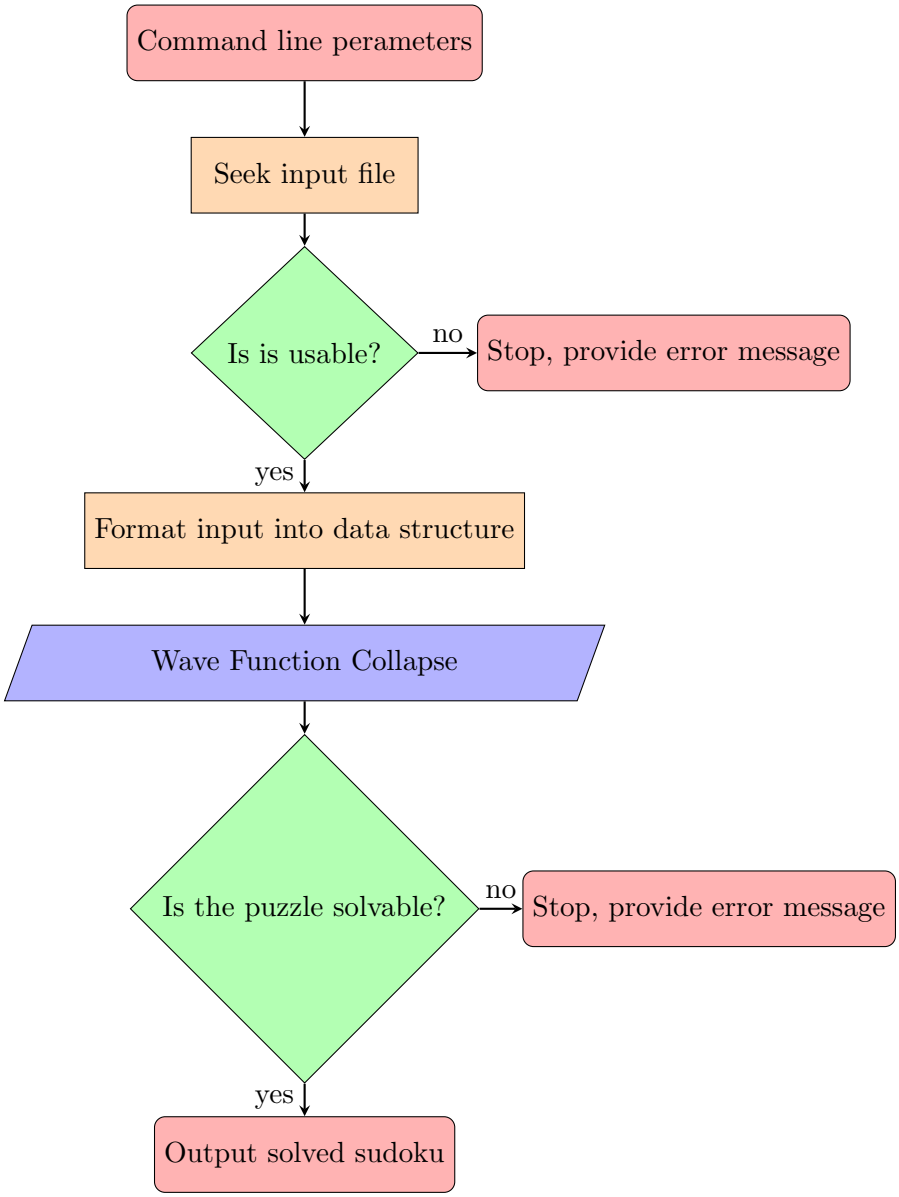
Wave function collapse is an algorithm that has a scary name, but is surprisingly simple, if resource intensive. The idea is that if an item has an unknown state (like a sudoku tile that isn't filled in), conceptually it can be considered in all of the states it could possibly be in. For exmpale, given a very simple psudo-sudoku board, only going up to 4 (as opposed to 9):

4	A	2	B
_	_	_	_
_	1	_	_
_	_	_	_

Consider A and B to be tiles that aren't filled in. If only the top row is considered, the wave function collapse algorithm would say that both A and B are in both a state of being 1 and a state of being 3, at the same time! But if the program then considers tiles outside the top row, it sees that in the second column, which A belongs to, there is a 1. Since A now cannot be $\{1, 3\}$, instead the set of all

its current possible states is {3}. Because of this, it's real state is known, and it "collapses" into a single state, namely being 3. Since A is now 3, the set of all of B's states is now simply {1}, meaning it too can collapse into a single state.

4 Large picture process



5 Why Rust/How Rust?

The Rust language is ideal for this project for many reasons. Primarily, the type system it offers, combined with the pattern matching it provides, allows for excellent error handling for avoiding issues at runtime. Secondly, similar to the C family, Rust is a relatively fast language, which is imperative for this project, due to the nature of the algorithm in use.
Because of the Rust compiler and the Rust Foundation's clear stance on what idiomatic Rust looks like, using the common naming conventions shouldn't be an issue. For example, variables in snake_case, UpperCamelCase for data structures, SCREAMING_SNAKE_CASE for statics, etc. If something isn't idiomatically named, the Rust compiler will throw a warning. As with most Rust projects, defining what pardigm is in use is difficult, as Rust provides optionality for everything from (something like) OOP to (something like) functional programming. This project will use mainly a OOP design pattern,

basing around data strucutres, however useful functional features such as closures are encouraged. Discussing the use of specific crates, command line input should be managed by the Clap crate, I/O should be managed by the Serde crate and whatever format crate is relevant (likely the CSV crate.) What is likely easiest to do, in concrete terms, is to load in a CSV to a struct containing a 2D Vector of an enum that has two states: u8 and a Vector of u8, and iterate over that using wave function collapse. Note that Serde will require a custom deserialization.