Scala Project Report

Group "🤵>🥐"
Achim Constantin-Marius, Oleksik Vlad-Andrei
vladao

IKT212
20th September 2024

Abstract

This work represents a functional, axiomatic system–inspired approach to solving instances of Tracks puzzles, implemented in Scala around a backtracking algorithm with strong validation rules in order to minimize the search time cost. While the worst-case time complexity for this approach is polynomial, in practice, this approach is most often feasible and exhibits subexponential growth in the time cost. The application was evaluated in an interactive environment and showed all empirical signs of correctness.

1 Introduction

The aim of this project is to design and implement, in the Scala programming language, a solver for the general case of the "Tracks" puzzle. The aforementioned puzzle is represented by a rectangular grid of cells, each either containing a track connecting precisely two edges of the given cell, or not having a track at all. Given at least the starting and ending point of a track not crossing itself and the total number of track cells on each line and column of the grid, the goal of the puzzle is to find a path formed by such contiguous tracks and following the above-mentioned constraints, given such a path exists and is unique. A more complete description has been established by [5].
As Scala, the language in which the solver is to be described, is predominantly a functional programming language, it stands out through the presence of features such as an expressive type system [4], higher-order functions and lambda abstractions [3], aimed at providing a functional way of describing an algorithm. Thus, besides the correctness and efficiency of finding a solution to the puzzle, embracing a functional paradigm rather than an imperative one represents a key aspect for the present project.

2 Solution

2.1 Problem modelling and data structures

In order to approach such a problem, the first aspect faced is that of defining a representation (equivalent to a recursive language [1]) for the puzzle so that it can be accepted and solved by a Turing machine in a finite time. Thus, one of the key aspects that influenced the procedure for solving a puzzle as described above was the way information regarding a certain puzzle is structured in the memory and handled by the solving algorithm.

2.1.1 Puzzles

The structural unit of this problem, on which functions operate in order to find a solution, is the concept of *Puzzle*.
A *Puzzle*, in the scope of our solution, represents the state of a rectangular grid containing cells that can, at a time, have various degrees of certainty of containing a track piece (as described under "Cells"), along with information about the counts of cells on each row and column, as well as information about the endpoints and grid size.
In the concrete Scala solution, this entity is represented as a `class` with the following structure:
The *Puzzle* is constructed from a `List` of `String`s, beginning with its dimensions on

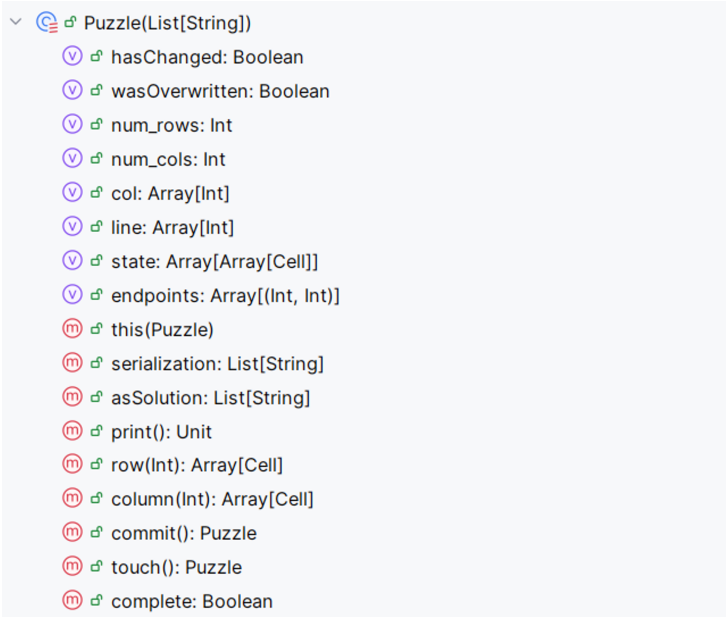

Figure 1: The structure of the Puzzle class.

the first line, continuing with a line containing the count of tracks on every column as a hint, then followed by each line of the puzzle, with tracks marked where they are certain and an "_" character to mark uncertain tracks. The "serialization" of a *Puzzle* represents the puzzle as a list of strings, using the same conventions as in the input. There is also a method for constructing a (deep-)copy of a puzzle.

As it can be seen, a *Puzzle* also has a set of methods described as propositions about it, such as it being complete. A relevant observation to make is that, under our representation of a puzzle as a grid with more or less information about its content, the particular case of a *complete* puzzle, one with no entropy left in the state of its cells and abiding by the rules of the Tracks puzzle is defined as a *solution*.
For the scope of this problem, we will say two *Puzzles* to be *equivalent* if and only if, under the rules (described in the following part of this section) of the Tracks puzzle, they can be reduced to the same *solution*.
Thus, by solving the puzzle, we understand traversing the puzzle space from equivalent puzzle to equivalent puzzle, minimizing the distance to a solution in this space.

2.1.2 Cells

Another aspect that is to be defined is the representation of a certain track piece (or the lack thereof) in a certain cell. For the purpose of this work, the model we have chosen for each cell on the grid specifies that each cell can either:

* be uncertain – in this case, the cell is not certain to either contain or not contain a track;
* be "empty" – in this case, no equivalent puzzle will contain a definite track in that cell;
* be "filled" – in this case, the solution contains a track portion in the cell, but it is not known which track, precisely;
* be "known" – in this case, the cell contains a track piece and it is known.

An observation that can be made is that, for the case of a known cell, the known track will connect two exactly two of the adjacent cells.
All this information regarding the cell will be referred to as the *state* of the cell. This state will be represented as an integer in the context of a *Cell*. The two least significant bits of the `state` will be encoding how much we know about the cell (`0b00` – uncertain; `0b01` – known; `0b10` – filled; `0b11` - empty), while the next 4 bits will each be used as a flag, each corresponding to one of the directions the track connects to, as depicted in Figure 2.

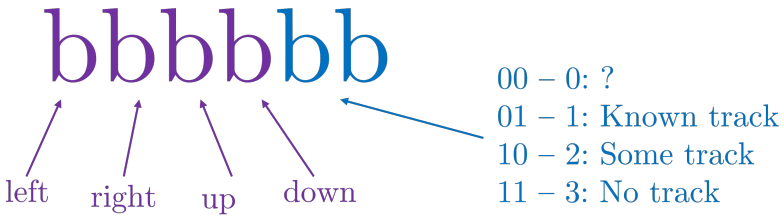

Figure 2: The memory map of a Cell state.

There are also numerous property-like methods for obtaining specific data about a certain cell (for instance, `goesUp` or `isTrackCertain`). At the level of a *Puzzle*, its state is represented as as collection of *Cell*s.

2.1.3 Inference Rules

In the subsection describing a *Puzzle*, the concept of *stepping through the puzzle space* from one equivalent puzzle to another was introduced. However, our model, as described above, did not feature a mapping from a certain puzzle to another, more complete one.
In the programming paradigm this work follows, the means of describing such a mapping in a language like Scala is represented by a function, taking a *Puzzle* as a parameter and returning another one in the space.
The precise functions that perform such mappings are described as consequences of the rules of Tracks. As such, an *Inference Rule* may be *applied* on a Puzzle, resulting an equivalent Puzzle, as presented in Figure 3. This "contract" that shall be followed by

```
abstract class InferenceRule {
  def apply(puzzle: Puzzle) : Puzzle
}
```

Figure 3: The contract enforced for all functionally-defined inference rules.

all inference rules is enforced in the `InferenceRule` abstract class, declaring a single pure method that has the structure of the mapping described above.
As such, chaining multiple inference rules (implemented as a composition of multiple rule-application functions) allows for the reachability of equivalent puzzles closer to the *solved* state, as depicted in Figure 4. A comprehensive list of inference rules and their

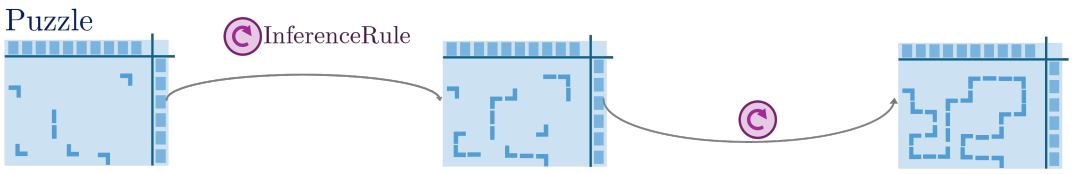

Figure 4: The application of inference rules.

brief description can be found in Appendix A.

2.1.4 Constraints

Another mapping we would like to define for this solution is represented by a *validation rule* or a *constraint*. Such a rule occurs as a consequence of the specifics of the Tracks game and declares any puzzle that does not satisfy a certain clause to be invalid (equivalent to an unsolvable puzzle). The integration of this structure into the solver algorithm will be discussed in section 2.2.
A comprehensive list of constraints and their description can be found in Appendix B.

2.2 Puzzle complexity and algorithms

In order to tackle this problem algorithmically, the first aspect that needs to be treated as part of the present project concerns studying the computational complexity of the Tracks problem.
Proving the Tracks puzzle as NP-complete [2], as with many other similar puzzles, would imply finding a solution with polynomial-time algorithm classes like Greedy is not possible and any set of rules described under 2.1.3. will be *incomplete*, meaning that they are limited by a boundary of reachability of puzzle states (a solution may thus not always be found just by chaining deductions. As such, any algorithm shall be capable of performing a superpolynomially-growing number of steps to guarantee being able to find a solution to a certain instance of the puzzle. As a consequence, we decided to use a backtracking approach for this problem.

2.2.1 Puzzle generation

Given a certain puzzle, all the rules are chained until there is no more information that can be extracted using the existing inference rule system. Once that point has been reached, this approach needs to add some information by making assumptions about certain cells. That means that the chain of equivalent puzzles "branches" into multiple versions of the puzzle, thus creating a "search" tree, where the algorithm is trying to reach the "solved" state.
One important observation to make is that the set of all the puzzles that only differ by one cell state covers the entire search tree for the general puzzle. Thus, there is no need to make assumptions about more than one cell at a time. Relying on these observations, we can use any strategy for selecting a cell not fully determined (i.e., the one with the most connections, the one that continues the path already formed etc.), to then try substituting it for every possible determined state it might have. The outline of the generation function can be found below.

```
def generate(puzzle: Puzzle) : List[Puzzle] = {
    var list : List[Puzzle] = Nil
    enumPossibilities(findCell(puzzle)).foldLeft(List[Puzzle]())((acc, cell
    )=>(
      {
        val newPuzzle = Puzzle(puzzle)
        newPuzzle.state(cell.row)(cell.col).state = cell.state
        acc :+ newPuzzle
      }
    ))
}
```

Listing 1: Generation function

2.2.2 Validation function

Since the time cost of searching through the whole possibility space (*bruteforcing* the problem) would be too high to handle practically, a significant emphasis will be put on minimizing the recursion depth by having a wide set of inference rules, and, more importantly, minimizing the recursion breadth by pruning as many branches as early on

as possible. For this, we have devised a set of constraints that are applied as part of getting a puzzle's validity. The code for the validation function is presented below:

```
def tracksValidate(puzzle: Puzzle, constraints: List[Constraint]) : Boolean
    = {
    constraints.forall((c: Constraint)=>c(puzzle))
}
```

Listing 2: Validation function

2.2.3 Backtracking function

Given the discussion above, the structure of the backtracking function is the "canonical" one – applying all the rules until the puzzle can no longer be simplified through inference alone, validate the state to make sure a solution can exist for this state, then generate a list the minimum amount of puzzles to cover the entire search tree and backtrack for every such puzzle in the list, as depicted in the figure below. The implementation of the

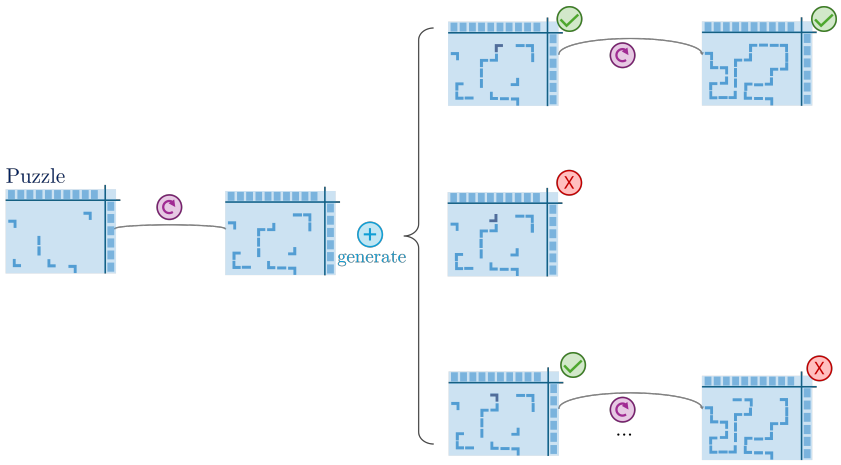

Figure 5: The backtracking control flow.

backtracking can be done in two quite distinct ways: *recursively*, using the program's call stack to store the state of the search, and *iteratively*, by implementing a stack of our own to store the state of the search, that is, all the general cases yet to be visited. Although the recursive definition is, syntactically, the more functional one, for the present work, the iterative implementation has been chosen, trading the simpler flow control for a data structure with fewer limitations, bringing more generality and reliability to the solution.

2.3 Solution architecture and design patterns

Since the program was designed with the benefits of the functional programming paradigm in mind, such as flexibility and more descriptivism (handing some of the control flow responsibility to the language implementation), the architecture of the program revolves around a `PuzzleSolver` app that, through the `PuzzleReaderWriter` class, handles reading the input data, then constructs a solving strategy with a backtracking backbone

and a variety of inference rules and constraints to then solve the puzzle and output it.
One of the key aspects of the design of this application, though, is that the solving strategy represents a *family* of functions, each produced by a "factory–function", depending on the specific set of validation and inference rules desired. This strategy for creating objects (functions) is equivalent to the *Builder* and *Factory method* creational patterns [3]. As such, any strategy can be created by simply passing the rule–objects as parameters to a builder–function. This allows for a very high degree of flexibility in describing new rules, as their definitions can exist independently of the solver, unless invoked, as shown, for brevity, for the validation scheme, in the code snippet below.
If the backtracking algorithm is the backbone of the solver application, its definition is contained in the rules, each encapsulated in an object inheriting from an abstract class/interface and introduced as a dynamic dependency through dependency injection.

```
def customTracksValidator(constraints: List[Constraint]) : (Puzzle =>
    Boolean) = {
   (puzzle: Puzzle) => tracksValidate(puzzle, constraints)
}
```

Listing 3: "Builder" for a validation function.

Furthermore, the outlining of general steps to follow in the backtracking function, without prescribing every operation, is resembling of the *Template method* behavioural pattern.

3 Correctness

3.1 Theoretical discussion

As the architecture of the implemented app, besides the assumption and backtracking steps, is based on an inference rule system to "advance" through the search space, the correctness of the algorithm relies mostly on the property of the inference system of it not being possible to deduce contradictory statements from the rules (the *consistency* of the system). Assuming the consistency of the system – empirically, no inconsistencies were observed in puzzles known to be solvable – any inconsistencies that appear in a puzzle-solving step can be pinned down to an invalid puzzle (or an invalid branch in the search tree), for which some error handling has been implemented.
Regarding the worst-case *efficiency* of the puzzle, the existence of an algorithm of a lower complexity class than that of the backtracking ($O(c^n)$ – exponential) would depend on the existence of a set of inference rules that is *complete* (can deduce the state of every cell for a puzzle with a single solution).
On the general case, however, as will be discussed in 3.3., the use of a *strong* validation function, of polynomial complexity, even with so-called "axioms" that are not necessarily independent of each other, will reduce the search space to a reasonable enough dimension for the problem to be practically solvable even for large input sizes.

3.2 Code analysis and programming paradigm

From a conceptual point of view, the implementation of the solving program has a functional description, as most of the code is handling the solving logic and the desired/de-

clared properties of the puzzles, instead of the solving control flow itself, at times left as an implementation detail. For the sake of brevity, below is presented a function checking for the presence of too little tracks being still possible to fill on a row or column.

```
object TooLittleRemainingTracksConstraint extends Constraint{
  def apply(puzzle: Puzzle): Boolean = {
    puzzle.line.zip(Array.range(0,puzzle.num_rows)).forall((total: Int,
    index: Int) => puzzle.row(index).count((c: Cell) => c.isTrackPossible) >
    = total)
      && puzzle.col.zip(Array.range(0,puzzle.num_cols)).forall((total: Int,
     index: Int) => puzzle.column(index).count((c: Cell) => c.
    isTrackPossible) >= total)
  }
}
```

Listing 4: Constraint to be applied to a puzzle.

As it can be seen the, code is more declarative, the focus being on the property of the *Puzzle* of not having too many impossible tracks, while the responsibility for performing the actual check are passed to the language.

3.3 Performance and testing

In order to ensure the correctness and the evaluate the efficiency of the solution implemented, a battery of unit tests were run on the program, covering multiple cases and input sizes. While the program only produced correct output for the instances tested, for a few cases, the time taken for producing the output was high.
Empirically, the most problematic cases in the tests were specifically the ones with areas with a high unknown track density and slithering tracks in the solution. The performance on these tests is also highly dependent on the generation strategy used in backtracking. For this reason, the best performance in practice in the testing environment was found for an approach using heuristics and a combination of generating functions.

4 Plan and Reflection

4.1 Planning and development timeline

The planning for this project spanned approximately five weeks, the first two weeks being centered, according to the initial plan, on understanding the problem and possible approaches, the next week seeing the translation of the gathered experience into a project architecture idea and starting the work on creating a base structure around which to place other entities, while the last weeks were focused on reaching the targeted reliability and improving performance.
The initially intended solver algorithm, implemented as a proof-of-concept under an imperative paradigm in C++ to review its efficiency, involved enumerating every possibility for every cell not initially fixed in the input data. However, this approach showed quick signs of inefficient behaviour, even for small instances of the problem. A subsequent proof-of-concept approach attempted to trace a continuous path from one endpoint, only evaluating three possible tracks at any point in the process. This lead to a somewhat

improved efficiency, but prompted a change in the plan for approaching this project.
With the realisation that the validation function should not merely eliminate extremely unreasonable puzzles, but instead it should try to eliminate a majority of options at any point, we decided upon a structure revolving around the backtracking function, but with nearly unlimited flexibility towards new rules for validation. This plan saw the even division of the rules we felt the need for among the team, leaving around 6 hours for rigorously describing a rule, implementing, testing and documenting it. The input and output was implemented in one day, as per the plan. The central element of the program was expected to take around three days, in which the generation, the validation, and the search were to be added, one by one. However, changing the backtracking code from recursive to iterative and experimenting with priority queues for possibly improving the general-case performance, as well as changing the generation function to prioritize different traits have taken significantly more time, which was thankfully accommodated for by streamlining the rule formulation process.

4.2 Reflection

Although some unforeseen turning points in the project planning did occur, we consider the prototyping phase to be of extreme importance for understanding the models one can construct for such problems and their limitations.
The approach based on encapsulating each rule and developing the system in parallel proved significant in organizing and streamlining the application development.

5 Summary

While this application does have some practical limitations, it is correct and efficient, its development showcasing the importance of the model and programming paradigm chosen, since some problem-solving processes favour an approach based on descriptivism, but perhaps more importantly, it illustrates the advantages of using a system of "strong" validation rules for problems that require backtracking, as the polynomial time cost of applying them brings little drawbacks while they promise a significant speedup by reducing the number of branches to visit per level of backtracking.
The modelling of the problem for this application will also represent a more versatile platform for translation into to other programming paradigms.

References

[1] Noam Chomsky. "On certain formal properties of grammars". In: *Information and Control* 2.2 (1959), pp. 137–167. ISSN: 0019-9958. DOI: https://doi.org/10.1016/S0019-9958(59)90362-6. URL: https://www.sciencedirect.com/science/article/pii/S0019995859903626.

[2] Computers and Intractability: A Guide to the Theory of NP-Completeness. *Programming in Scala*. first. W. H. Freeman and Company, 1979. ISBN: 9780716710455.

[3] Lex Spoon Martin Odersky and Bill Venners. *Programming in Scala*. fourth. Artima Inc., 2020, pp. 33–34, 197. ISBN: 9780981531618.

[4] *Scala docs*. URL: https://docs.scala-lang.org/scala3/book/types-inferred.html.

[5] Kevin Stone. *Brainbashers: Tracks*. URL: https://www.brainbashers.com/showtracks.asp.

A List of inference rules

Rule formal name	Description
AdjacencyRule	*All uncertain cells with incoming tracks from knwon cells become "filled".*
BottleneckRule	*A row/column that is filled by a continuous track starting from one of the endpoints cannot be crossed again.*
LessThanTwoNeighboursRule	*Any cell with less than two possible neighbours is "empty".*
NoTracksRemainingRule	*Once all the filled cells from a row/column have been identified, the rest are marked as "empty".*
OnlyTracksRemainingRule	*Once all the empty cells from a row/column have been identified, the rest are marked as "filled".*
OnlyTwoGoodNeighboursRule	*A filled cell which has only two neighbours that are possible, its track will connect to those specific neighbours.*
OnlyTwoGoodConnectingNeighboursRule	*Any cell that has precisely two known track portions connecting to it has its track connect to those specific neighbours.*
RemoteAreaRule	*A cell in an area bordered only by cells with no tracks is empty.*

B List of constraints

Constraint formal name	Description
AdjacencyConstraint	*The existence of a cell with an incoming track and without a matching outgoing track deems the puzzle invalid.*
MoreTracksThanPossibleConstraint	*A row/column with more tracks than specified by the hint deems the puzzle invalid.*
NoContradictionConstraint	*Any cell state that can have two different values deduced using the rule system deems the puzzle invalid.*
NoLoopsConstraint	*The existence of a looping train track deems the puzzle invalid.*
NonIsolationConstraint	*A portion of tracks isolated from both endpoints by another track portion deems the puzzle invalid.*
NoThreesomesConstraint	*A cell with more than two connecting tracks deems the puzzle invalid.*
OffByOneConstraint	*A row/column that can only accept an even number of tracks being added to it but is missing one track deems the puzzle invalid.*
TooLittleRemainingTrackConstraint	*A row/column with two little possible tracks deems the puzzle invalid.*