University of Agder Norway

Prolog Project Report

Group - The MVBs -
Benjamin Kelly, Marius Achim, Vlad Oleksik
- vladao@uia.no -

IKT212
01.12.2024

Abstract

This assignment involves developing a Tracks Puzzle solver written using the Prolog programming language. Unsolved puzzles are provided to the program as strictly formatted, ASCII-encoded text files. Upon being solved, the solution is written into a text file of identical format. The solving approach closely resembles that of the Scala implementation, applying inference rules related to a cell's surroundings from one endpoint to another as a way to describe moves between states in a dynamic problem. Where Prolog's code stands out is in its inbuilt ability to emulate logical reasoning. Instead of explicitly formulating backtracking mechanisms, Prolog employs them out-of-the-box.
In terms of performance, the final version of the solver successfully solved 63 out of the 101 puzzles provided by the teacher, with an average execution time of less than 1 min. The remaining 38 puzzles exceeded the 5-minute execution limit and could not be solved within the allowed time.

Contents

1 Introduction

Continuing on implementing Tracks solvers in different languages, a Prolog-based solver was developed. It incorporates many constraints and mechanisms of earlier projects, now expressed through Prolog's facts and rules.
As outlined in the previous two reports, tracks puzzles consist of a rectangular grid of cells, with rows and columns labelled by numbers indicating the required number of tracks to be lain in their respective axes. The goal is to construct a continuous track that connects to endpoints while avoiding collisions, junctions and loops. Due to the well-defined nature of the puzzle's specifications, the search space can be severely restricted, corresponding to what, according to the task description, is a unique solution to each puzzle.

2 Solution

The following sections delve into the overall structure, design rationale, strengths, and limitations of the Prolog solver.

2.1 Puzzle representation

To reason about the puzzle's board, it must be stored in a predefined or at least predictable data format. For this purpose, a complex structure was defined as follows:

```
tracks(W,H,Hc,Hl,C).
```

Here, variables W and H represent the board's width and height, Hc and Hl are lists containing column and line hints respectively, and C holds a list of cells. For convenience, each cell is assigned three atoms, one describing its state and two corresponding to its column and row indices on the board.
The state of a cell is represented by one of the following atoms/facts:

- _: An empty cell whose state is to be inferred by Prolog.
- *imposs*: A cell where placing a track is impossible.
- *cert(some)*: A cell that must contain a track.
- *cert(horiz)*: A horizontal track.
- *cert(vert)*: A vertical track.
- *cert(nw)*: A track linking a north to a west-facing neighbour.
- *cert(ne)*: A track linking a north to a east-facing neighbour.
- *cert(sw)*: A track linking a south to a west-facing neighbour.
- *cert(se)*: A track linking a south to a east-facing neighbour.

One of the crucial advantages of this representation is the ability to represent any amount of information known at a time about a cell, without having to employ special cases. Thus, all the tracks are represented as varieties of a cell "certain" to contain a track piece. As such, any amount of information can be added at any time, preventing loss of information, as well as incompatible states. Any incompatibility (i.e., a certain cell that can be shown to be impossible) will not be able to be represented and will imply failing the current branch natively, though Prolog's pattern matcher.

2.2 File workflow

The process of handling puzzle files begins by defining desired input and output files. These are passed on to the solver over command line arguments. Following that, input and output file streams are opened. The first step involves extracting the number of puzzles from the input file, which is indicated on the first line before the prefix "puzzle". Assuming all puzzles are solvable, this number is written to the output file following the prefix "puzzles".
Subsequently each puzzle is read, solved and appended to that file. To know how often the puzzle reading process should be executed, the initial number of puzzles gets decremented recursively until 0 is reached.
Finally, both input and output file streams are closed.

2.2.1 Reading of Individual Puzzles

The program follows a series of steps to interpret a single puzzle.
First, its width and height are determined by the first line, which should adhere to the format "size *width*x*height*". Subsequently, having skipped over a new-line, all column hints are gathered up by reading *width* number of integers. The final values to determine are the cells and the line hints. These are read in order from left to right and top to bottom. Cell states are converted from Unicode codes to atoms as described in section 2.1. In order to accommodate for a slight difference in format across the testing data, namely, some files containing a trailing space after some lines, a predicate had to be designed to process the data accordingly upon reading.

2.3 Code components

For legibility's sake, aspects of the code are addressed in categories of: Counters, Getters & Setters, Validation mechanisms and Solving flow.

2.3.1 Counters

Despite their simplicity, counters are fundamental for both validation and solving. They collect all cells in a given axis into a list and then recursively iterate over the head while incrementing a accumulator whenever the current cell satisfies a given predicate. Just like other kinds of predicates that are inscribed, both from a logical and from a functional point of view, as observers, these ones are designed to evaluate as true only for cases where certain properties are satisfied (in terms of the information available at a certain intermediate stage). Since the representation employed for this solver natively modeled uncertainty as a placeholder to be filled by certain atoms, an aspect that is crucial to the strategy is maintaining the time complexity of evaluating these predicates. As such, some features of Prolog's syntax have been used to enable the evaluation of certain statements without forcing bindings over multiple variables at a time.

2.3.2 Getters & Setters

Since cells are stored in a contiguous single dimensioned list, accessing and modifying the board's state involves matching row and column coordinates through Prolog's inbuilt *member* rule as shown below:

```
getCell(I, J, L, C) :- member(cell(I,J,X),L), C=cell(I,J,X).
setCell(I, J, T, State) :- T=tracks(_,_,_,_,C), member(cell(I,J,X),C), X=State.
```

While this single-dimensional list effectively contains all cells of the board, it is not the most efficient structure for accessing individual cells. The time required for access increases with the distance of the cell from the top-left corner. This inefficiency is often mitigated by the *getLine* and *getCol* rules, which memorize all cells in an axis at a given position at once, using unification, for future use. These procedures themselves use a single pass over the cells to obtain the desired line/column.

2.3.3 Solving mechanisms & algorithm

In view of the fact that Prolog's strength lies in its ability to pattern match and the board's uncertain states are represented as variables, the solving-strategy employed is to formulate rules that unify said uncertain states to satisfy validation steps.
To achieve this, the solver iterates over each board cell, isolates its four neighbours, and applies inference rules to the central cell based on its relationship to these neighbours. Some rules are applied over the entire row/column a certain cell is part of. In this case, the rules get applied iterating through rows and columns, employing a single memory access for that specific application of the inference, thus reducing the program's compute- and memory-footprint.
Cell-isolation is handled by a specialized version of *getCell*, namely *getNeighbours*. By binding neighbouring cells through this rule before applying inference rules reduces redundant board access and decouples data access from inference logic.
Because only a subset of cells will satisfy specific inference rules, each rule includes a fall-through clause that performs identity matching between the input and output board. This ensures no modifications are made to cells that do not meet the conditions of the rule. A solution is considered valid once all axis hints are satisfied. This validation occurs after inference rules have been applied to every cell on the board.
Since solving this puzzle through the repeated application of this (finite) set of rules would imply the existence of a polynomial-time algorithm for solving Tracks, it is immediately apparent that there exists a point where these rules can no longer be applied to obtain any extra information about a certain instance of the puzzle.
Thus, in the case of an unsolved puzzle, the solver always proceeds to the next known cell in the direction the track is pointing until an unknown is reached, at which point a potential state i.e. a guess is generated. This represents the core element of the backtracking the

Prolog natively employs to reach a solution.
After all the guesses have been proven incorrect, the entire level of the current predicate fails, the interpreter thus backtracking to the most recent past guess implied.
Thus, this would roughly describe the structure of the search space in a dynamic problem, in which a move between two states is conditioned by certain inference rules that can provide guarantees that the two states lead to the same (unique) solution.
In this search space, illustrated by the image below, the breadth of the search can be observed to be very high, even as compared to other types of similar puzzles. As such, a logical system for restricting this search space in view of the most recent information about the particular state assumed at a certain point. This aspect is further discussed in the following section.

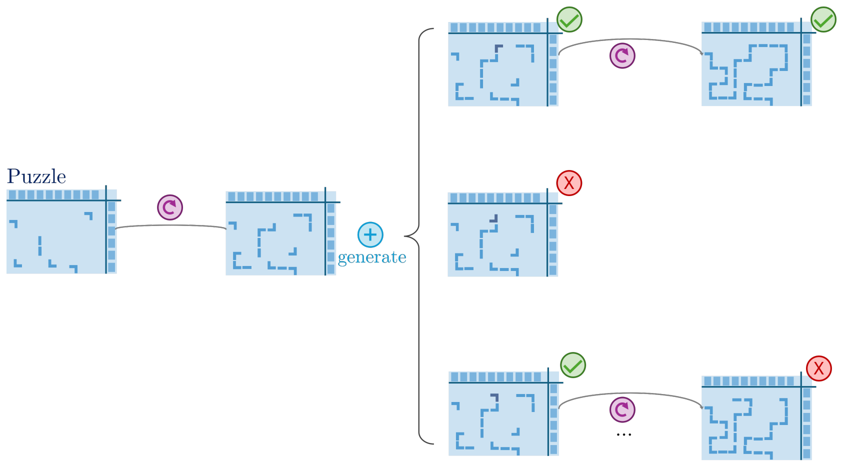

Figure 1: Dynamic traversal of the search space.

2.3.4 Validation mechanisms

Both before and after a puzzle solving attempt the board must undergo rigorous checks. The preceding step ensures no processing of demonstrably unsolvable boards will be done by simply allowing only those, whose axes don't exceed their hints. The succeeding step fails if any one of the axis hints aren't met since failing any of the rules equates to having an invalid state.
Important to note is that during cell checks a double negation is employed as shown below:

```
not(not(CellState = cert(horiz)))
```

This prevents Prolog from assuming predicates target values are that of unbound cells, thus bypassing the very system put in place for reducing the average time complexity, forcing an exponential slowdown. It is to note that these statements do not influence the result from a logical point of view, but instead only enable the computation to be done effectively.

3 Correctness

The subsequent sections outline quality assurance in respect to the project's requirements, including tests and empirical evidence of correctness.

3.1 Tests

The solver's capabilities were tested in a teacher-provided Bamboo validation environment, where it successfully solved 63 out of 101 puzzles with an average execution time below 1 min. The remaining puzzles failed due to timing out after exceeding the 5-minute execution limit. There are two suspected causes of these failures. The first is the time-consuming process of cell retrieval due to the underlying state representation. The second is the possibility that the search space is too broad, preventing the puzzles from being solved within an acceptable time-frame. Introducing additional rules and/or cuts could help mitigate this issue.

3.2 Logical programming and architecture

This section will briefly present the features of the current work in a context related to the architecture of the solver solution and the style of programming employed in this respect.
A first argument for the logical programming used in this project is represented by the Prolog language itself, featuring very few language elements with associated side-effects. Although, as described in the previous chapters of this report, such features as the "Cut" operator have been used, the context is overwhelmingly one in which it does not affect the result of the computation, but it just declares certain properties that make some evaluations computationally redundant.
One context in which this operator has been used and has effects beyond logic is represented by the predicate used for encoding a certain cell state as a character for printing. In this context, the purpose of such an operator is to prevent the pattern matcher from assuming certain yet unknown cells in the case in which printing an intermediate state is required for debugging purposes. Still, this use can be neglected, since the purpose of a predicate used for printing is, by definition, a side effect itself. Some assert statements are used as well, although their purpose is simply to speed up passing certain data through precomputation, such as in the case of endpoints. This use does not affect the logical flow of the program.
As to the architecture of the solver, predicates are described in a modular manner, providing a robust structure and yet, relatively little redundancy. The addition of stricter rules for restraining the search space can be done without disturbing the structure of the program.

4 Plan and Reflection

Table 1 presents the finalized task schedule, detailing the estimated and actual durations for each task including task allocation.

Table 1: Task schedule

Name	**Estimated (h)**	**Actual (h)**	**Responsible**
Code representation & reader of Puzzle	2	4	Vlad, Ben, Marius
Code validation logic	10	6	Vlad, Ben, Marius
Code solving mechanisms / inference rules	15	20	Vlad, Ben
Document Abstract and Introduction	1	2	Marius
Document Solution	8	5.5	Vlad, Ben
Document Tests	1	0.5	Vlad, Ben
Document Summary	0.5	0.5	Vlad, Ben, Marius

Prolog, being one of the more demanding programming languages in this course, required a significant investment of time to become familiar with it. However, once the steep learning curve could be managed, we eventually found it to be more comprehensible than ACT-ONE. Since the earlier two reports covered Tracks puzzles in exhaustive detail and our solution being a derivative of those projects, there was comparatively little new content to write about. This saved us time to focus on implementing and refining the solver instead of dedicating it to documentation.

5 Summary & Reflection

This work has presented a logic-based approach to representing and solving instances of the Tracks puzzle. The Prolog-based solver outperforms the previous two solutions in terms of speed while offering greater flexibility regarding the board's dimensions. It does struggle with some puzzles due to the larger search space they pose but, nonetheless, the implementation performs well on a majority of the puzzles in the target-range. Furthermore, even for some of the largest puzzles, the artifacts of the testing campaign show a correct file, which indicates a timeout of the order of magnitude of the time needed to solve the puzzle, implying this strategy is scalable enough and is more sensitive to specific patterns troubling a solver rather than exponentially depending on the raw size of the puzzle.

While Prolog is designed to solve problems using (raw) backtracking, this result goes to showcase the importance of restraining the search space through pruning and constraint propagation, be it language-defined or explicit, along with the relevance of the dynamic problem model, based on constrained moves through a space, in solving problems requiring an exponential number of steps. This model has been emulated in the Scala solver implemented in the past, despite us lacking a formal understanding of this design, obtaining promising results.

Further improvements such as additional rules and more cuts could increase performance even further. A list of the rules and constraints used to traverse this search space will be presented as an Appendix to this work.

A List of inference rules

Rule formal name	Description
AdjacencyRule	*All uncertain cells with incoming tracks from kn-won 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.*

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.*
NoLoopsConstraint	*The existence of a looping train track deems the puzzle invalid.*
TooLittleRemainingTrackConstraint	*A row/column with two little possible tracks deems the puzzle invalid.*
No contradictions	*Any cell state that can have two different values deduced using the rule system deems the puzzle invalid. Previously an explicit constraint, now handled natively, entirely through the representation of the puzzle.*