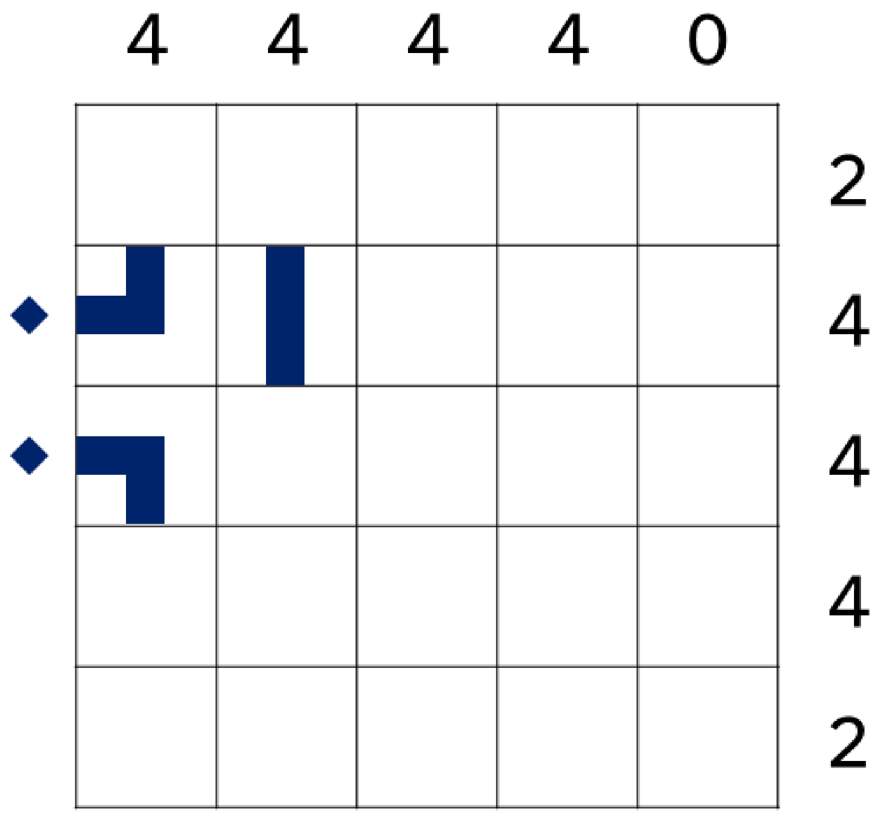
Tracks: A Programming Paradigms Odyssey

Benjamin Kelly | Marius Achim | Vlad Oleksik

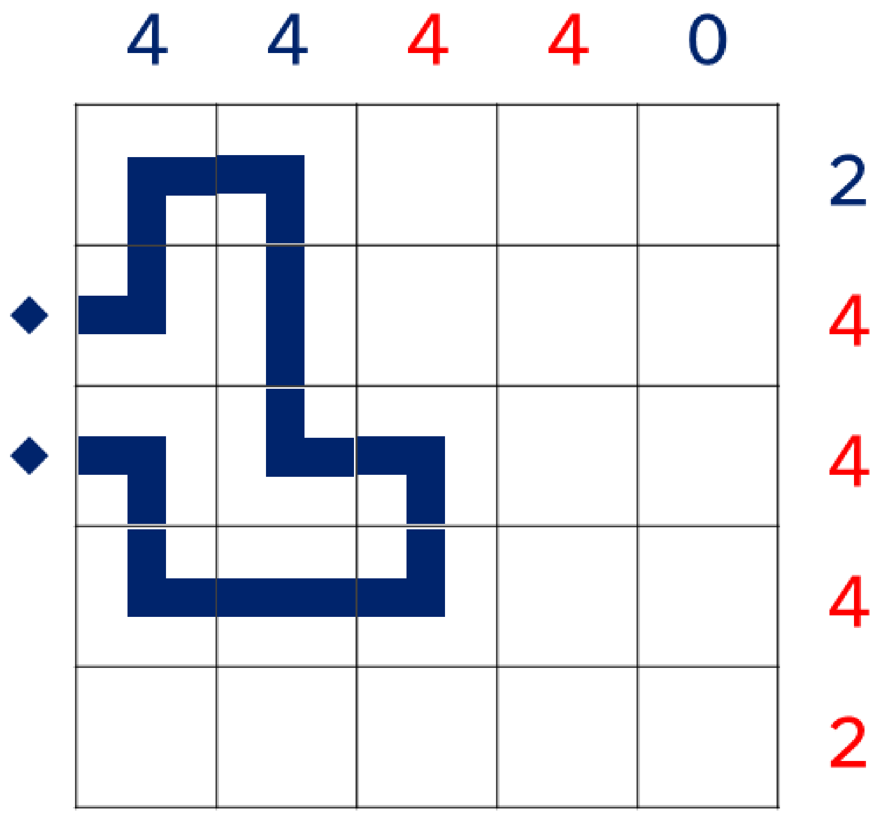
IKT212 – Concepts of Programming Languages

The problem



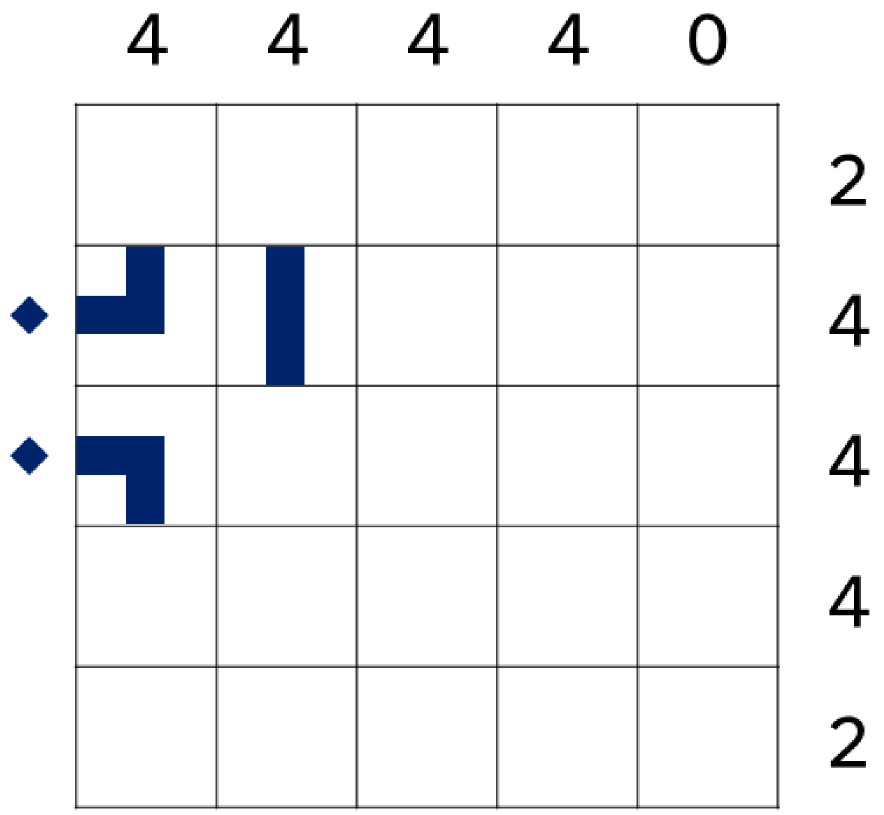
- An NxM grid of cells that can contain track pieces.
- At least a start point and an end point given.

The problem



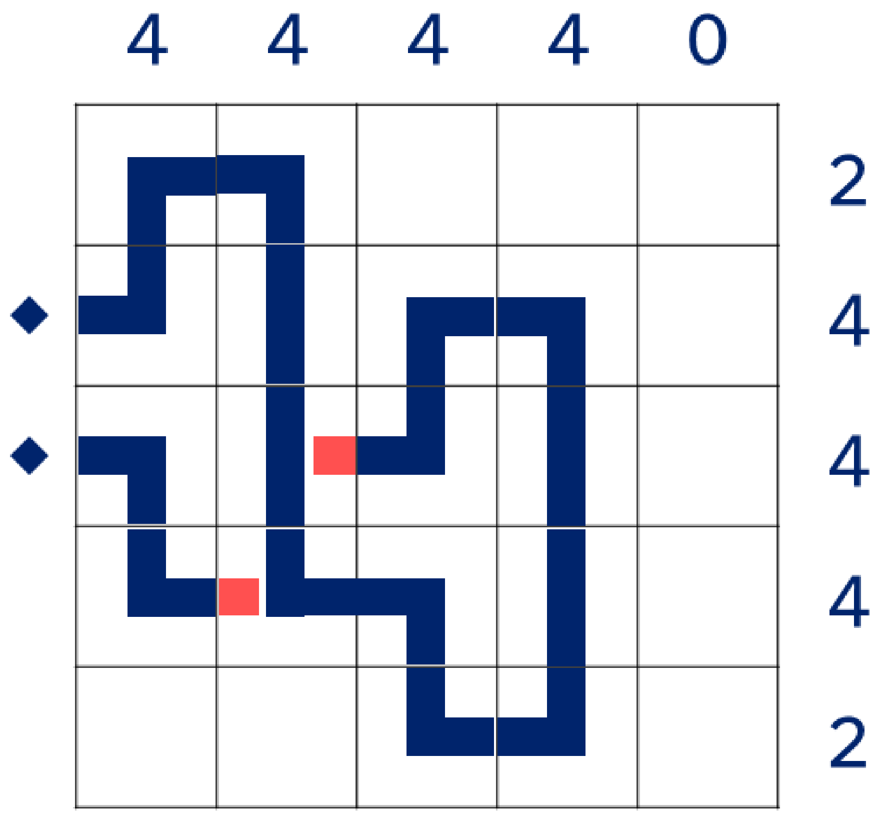
- An NxM grid of cells that can contain track pieces.
- At least a start point and an end point given.

The problem



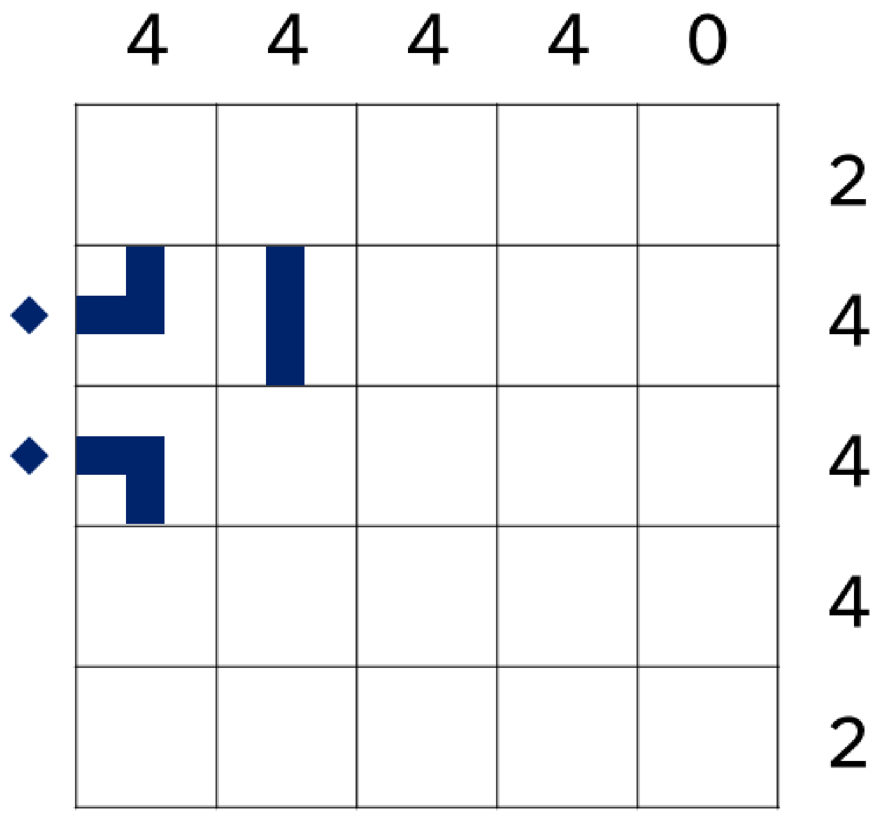
- An NxM grid of cells that can contain track pieces.
- At least a start point and an end point given.
- The number of tracks in the path connecting the endpoints must have, on each row/column, exactly the number stated by the respective hint.

The problem



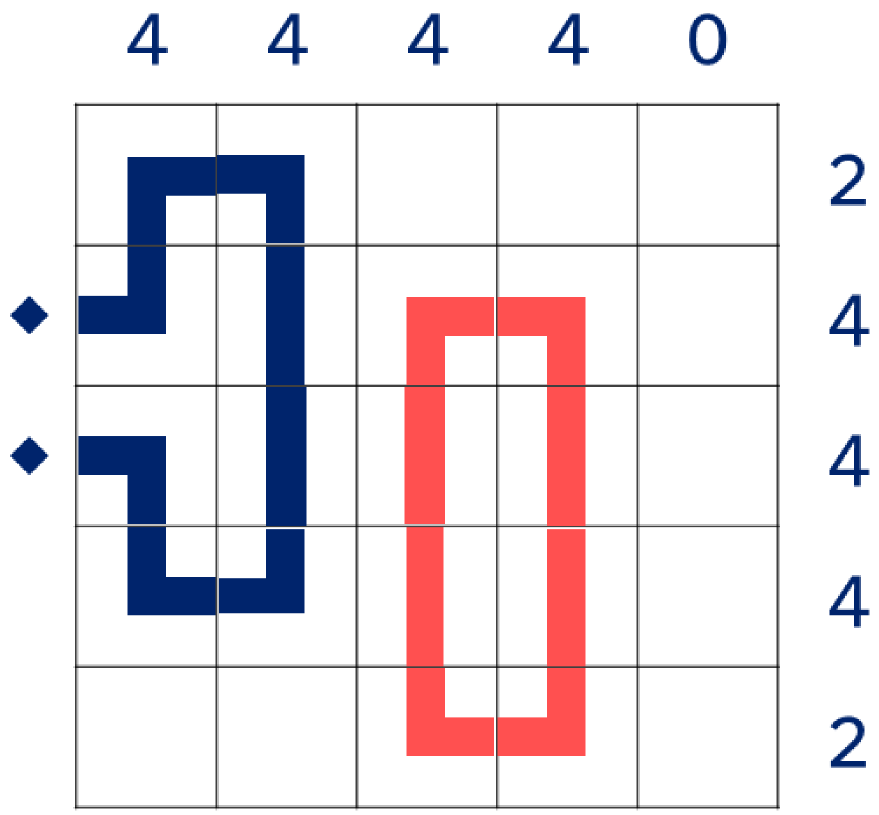
- An NxM grid of cells that can contain track pieces.
- At least a start point and an end point given.
- The number of tracks in the path connecting the endpoints must have, on each row/column, exactly the number stated by the respective hint.

The problem



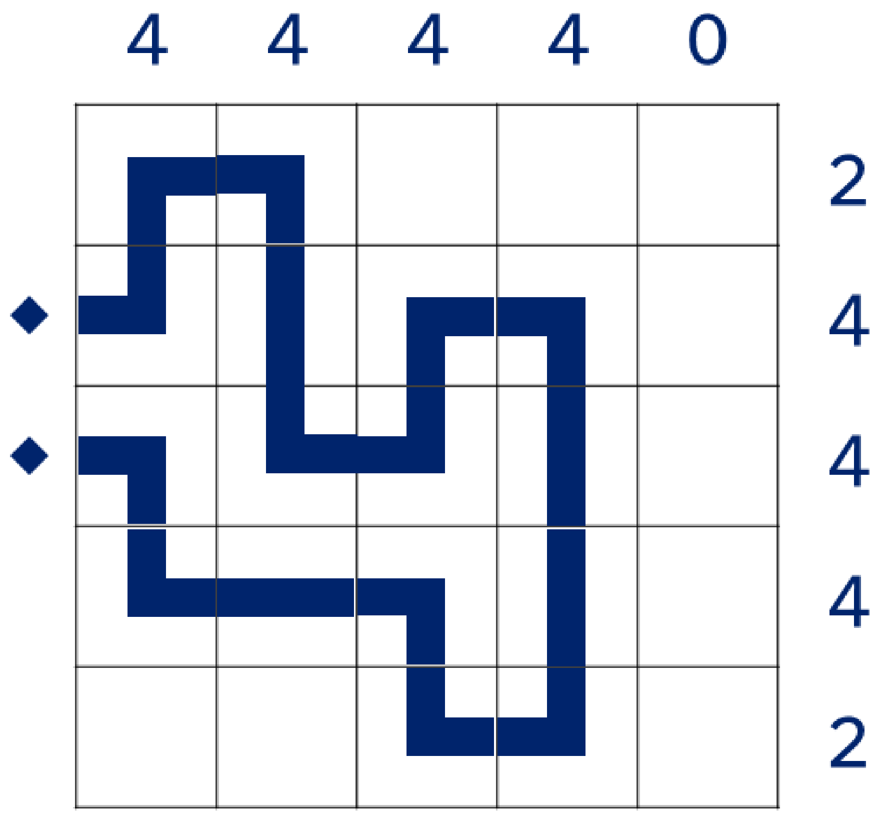
- An NxM grid of cells that can contain track pieces.
- At least a start point and an end point given.
- The number of tracks in the path connecting the endpoints must have, on each row/column, exactly the number stated by the respective hint.
- There can be no over- or underpasses, no T-shaped tracks or other intersections.

The problem



- An NxM grid of cells that can contain track pieces.
- At least a start point and an end point given.
- The number of tracks in the path connecting the endpoints must have, on each row/column, exactly the number stated by the respective hint.
- There can be no over- or underpasses, no T-shaped tracks or other intersections.

The problem



- An NxM grid of cells that can contain track pieces.
- At least a start point and an end point given.
- The number of tracks in the path connecting the endpoints must have, on each row/column, exactly the number stated by the respective hint.
- There can be no over- or underpasses, no T-shaped tracks or other intersections.
- There can be no loops of track.

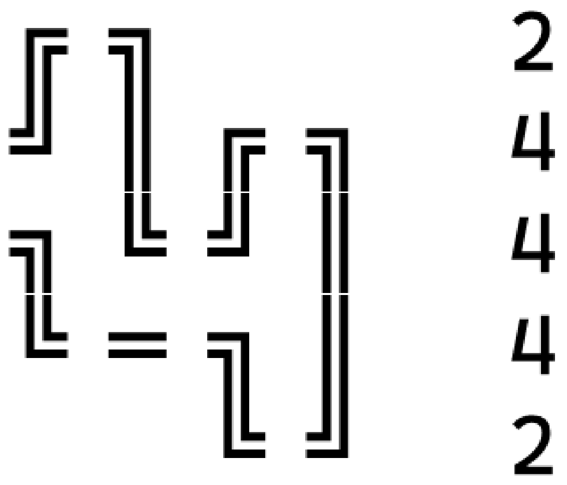
The problem – formal statement

```
puzzles 1
size 5x5
4 4 4 4 0
_ _ _ _ _ 2
╝ ║ _ _ _ 4
╗ _ _ _ _ 4
_ _ _ _ _ 4
_ _ _ _ _ 2
```

- An input file containing, on the first line "puzzles", followed by the number of puzzles to be solved;
- For each puzzle, a line containing "size", followed by its dimensions;
- A new line follows, having an integer for each column, indicating the total number of tracks for that column;
- A line for every row in the puzzle, having one of the following track characters: "═", "║", "╔", "╗", "╚", "╝" or "_" for an unknown cell, followed by a space;
- Each line is terminated after its corresponding hint.

The problem – formal statement

```
puzzles 1
size 5x5
4 4 4 4 0
```



- The output file structure is similar to the input.
- It must represent the solution to the puzzle. For an empty cell, it must write a space at the corresponding position.

Constraints:

- Each puzzle is guaranteed to have exactly one solution
- Maximum puzzle size: 58x50
- Time limit: 5 mins.

Part I: Scala

Functional programming

The Scala Language

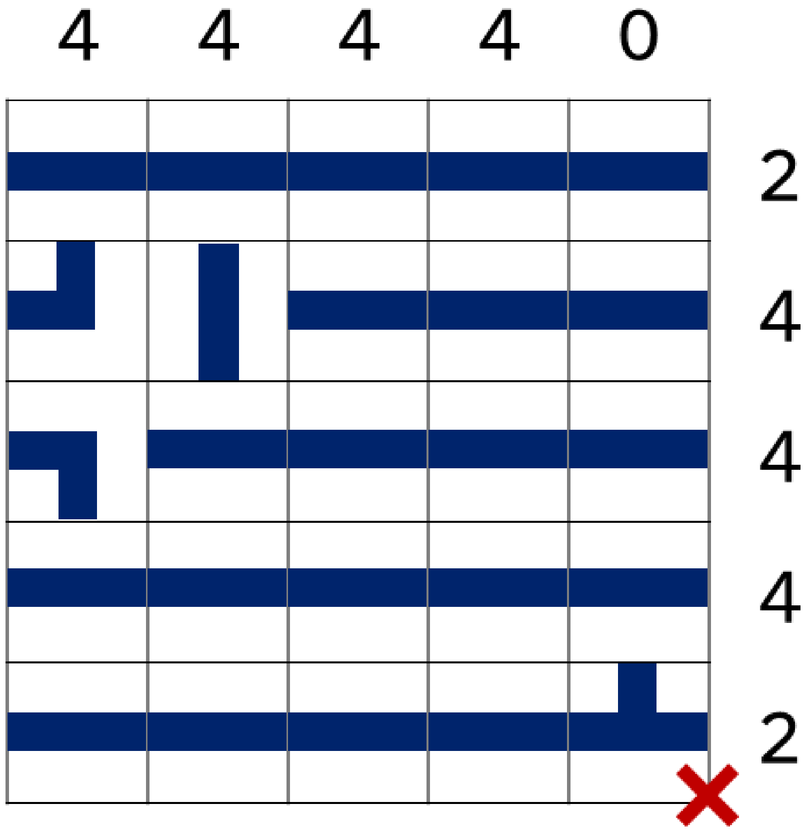
- A functional programming language based on Java
- Has numerous features from various programming paradigms

Rough solution design model:

Defining a function that takes an unsolved puzzle as a parameter and returns the solved puzzle, relying on the composition of other functions, each representing a step/case in the solving process.

Approach 1

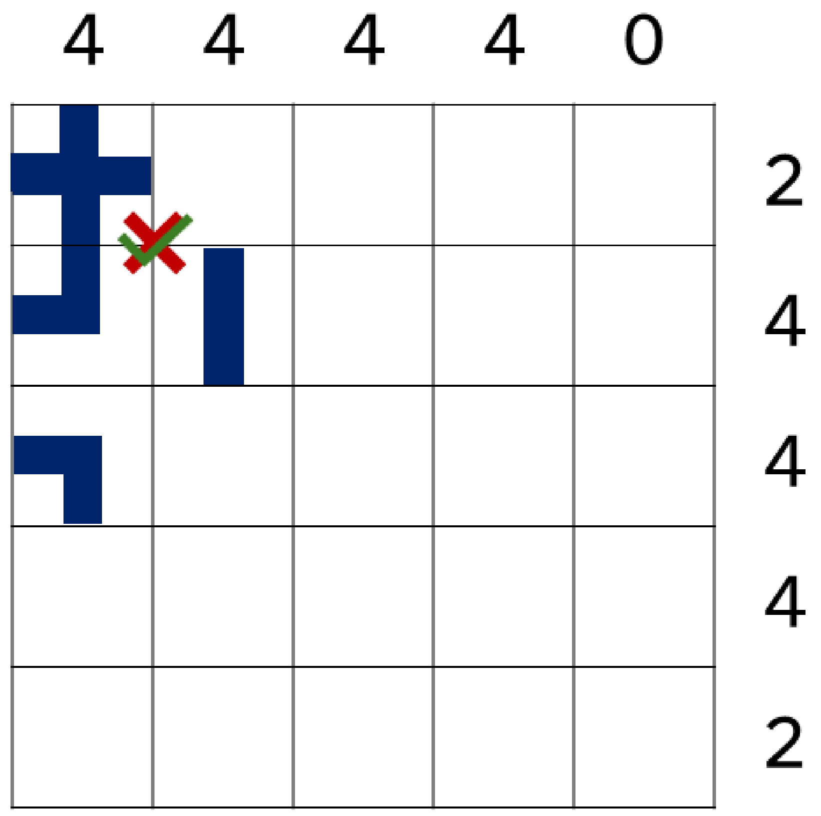
- "Guessing all the possibilities and seeing which is right."



$$solved(Puzzle, i, j) = \begin{cases} \begin{matrix} solved\big(placeAt(i, j+1, Puzzle, =)\big) \\ \cdots \\ solved\big(placeAt(i, j+1, Puzzle, ||)\big) \end{matrix}, & j < W \\ \begin{matrix} solved\big(placeAt(i+1, 1, Puzzle, =)\big) \\ \cdots \\ solved\big(placeAt(i+1, 1, Puzzle, ||)\big) \end{matrix}, & j = W, i < H \\ check(Puzzle), i = H, j = W \end{cases}$$

Approach 2

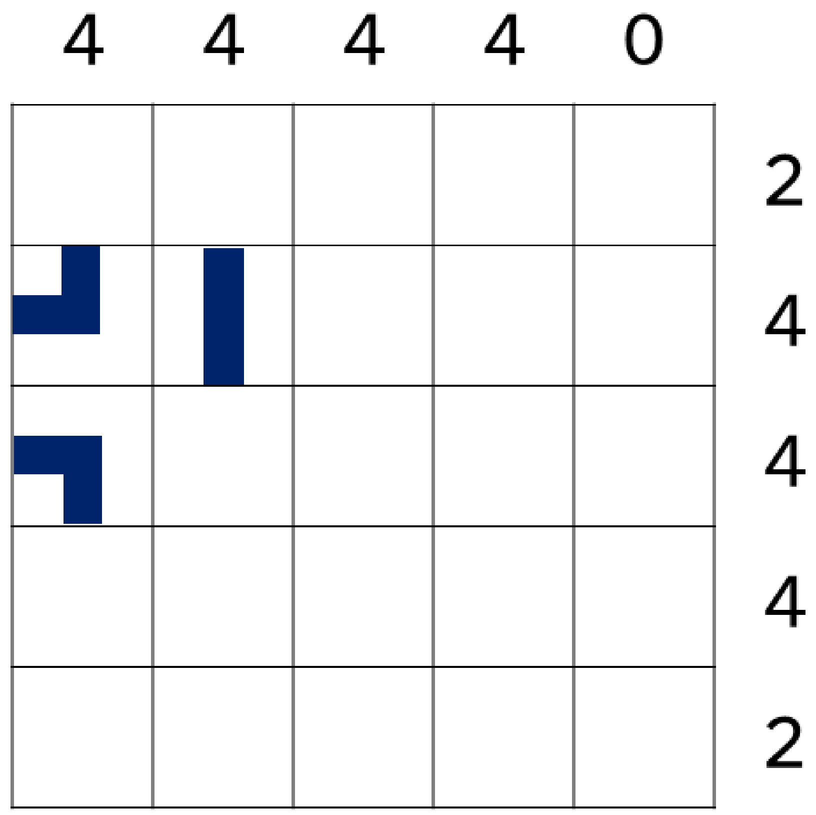
- **Stopping whenever what we have is clearly wrong.**



$$solved(Puzzle, i, j) = \begin{cases} \begin{matrix} solved\big(placeAt(i, j+1, Puzzle, =)\big) \\ \cdots \\ solved\big(placeAt(i, j+1, Puzzle, ||)\big) \end{matrix} & , j < W, checked(Puzzle) = 1 \\ \begin{matrix} solved\big(placeAt(i+1, 1, Puzzle, =)\big) \\ \cdots \\ solved\big(placeAt(i+1, 1, Puzzle, ||)\big) \end{matrix} & , j = W, i < H, checked(Puzzle) = 1 \\ check(Puzzle), i = H, j = W & \end{cases}$$

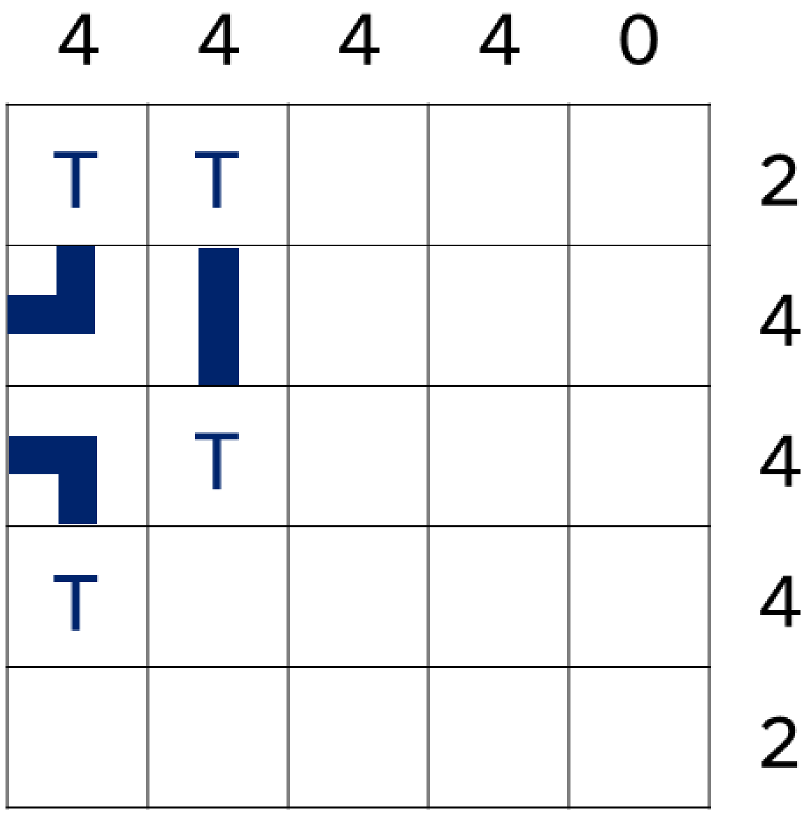
Approach 3

- Replacing as much guesswork as possible with deductions



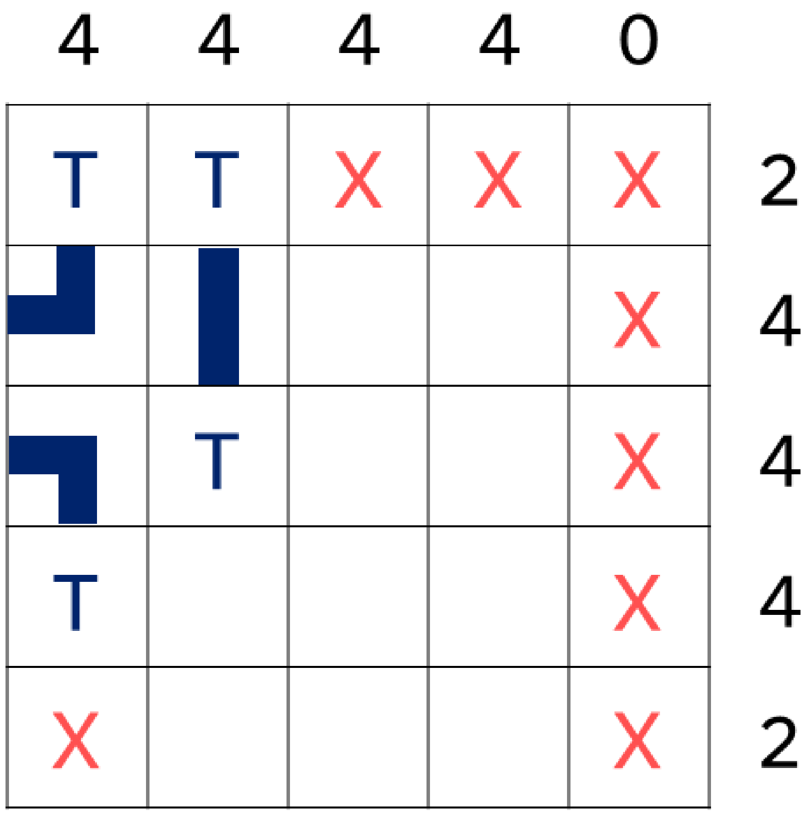
$$solved(Puzzle,i,j) = \begin{cases} \begin{matrix} solved(deduce(placeAt(i,j+1,Puzzle,=))) \\ \cdots \\ solved(deduce(placeAt(i,j+1,Puzzle,||))) \end{matrix} & ,j < W, checked(Puzzle) = 1 \\ \begin{matrix} solved(deduce(placeAt(i+1,1,Puzzle,=))) \\ \cdots \\ solved(deduce(placeAt(i+1,1,Puzzle,||)) \end{matrix} & ,j = W, i < H, checked(Puzzle) = 1 \\ check(Puzzle), i = H, j = W & \end{cases}$$

Deductions – Implied cells



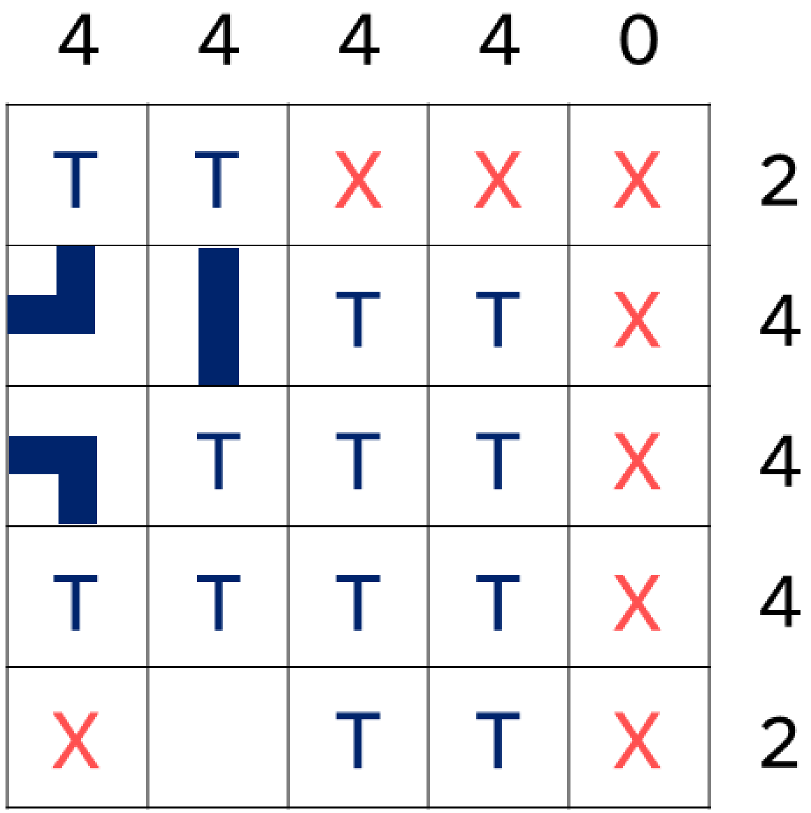
- Cells with incoming tracks are sure to have tracks themselves.

Deductions – Rows and columns



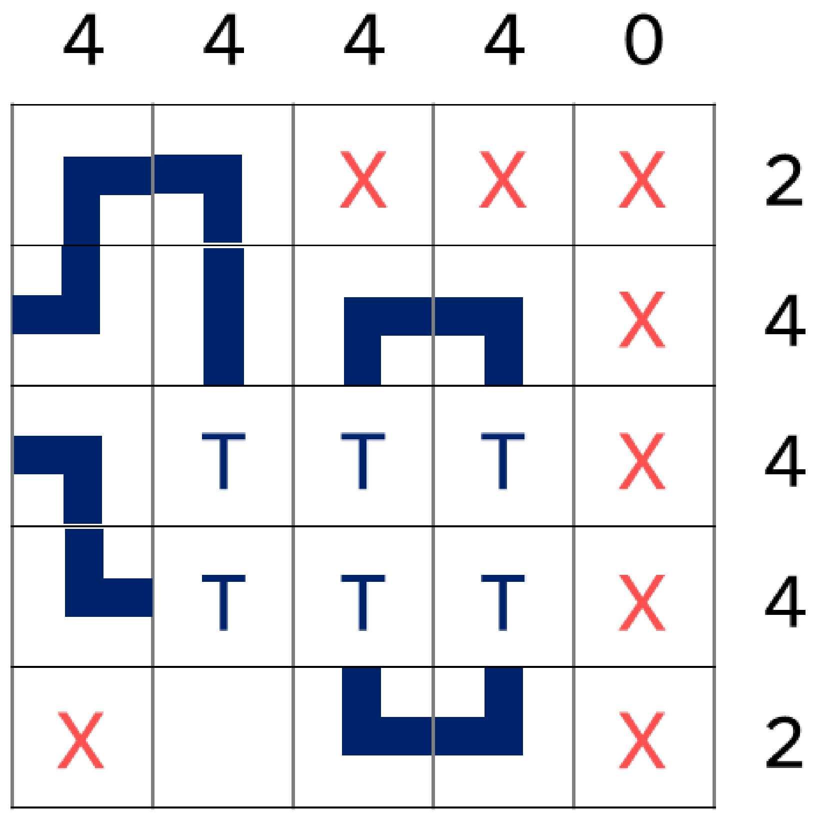
- Rows/columns with as many certain cells as the hint specifies will not contain any other tracks.

Deductions – Rows and columns



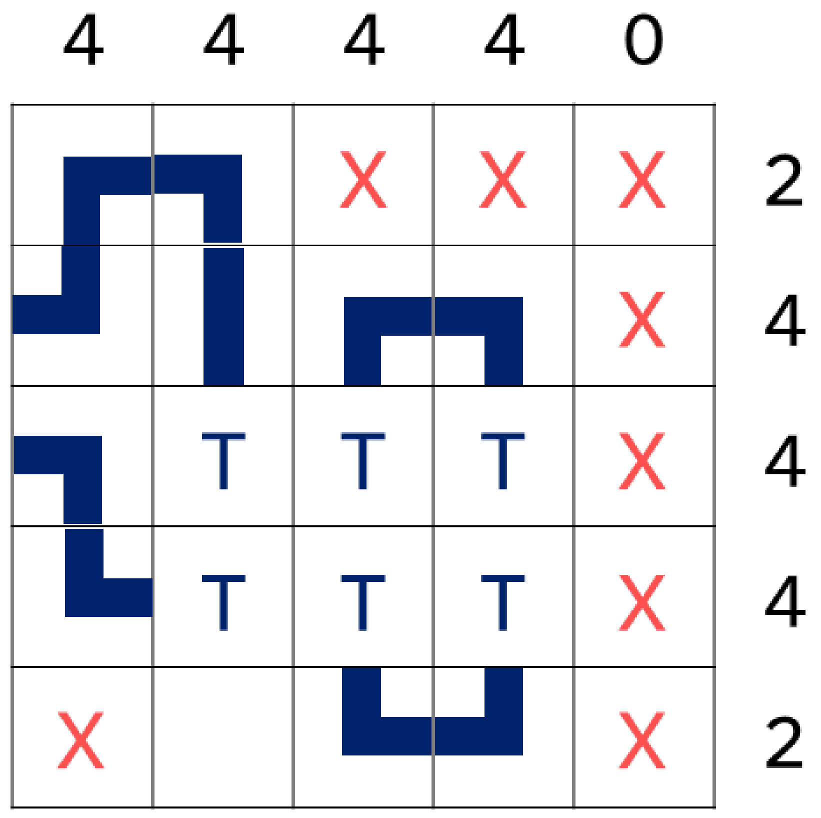
- Rows/columns with as many certain cells as the hint specifies will not contain any other tracks.
- Rows/columns that already have as many impossible cells as it is possible without contradicting the hint will not contain any other empty cells.

Deductions – Tracks



- Cells that have a track and only two possible connections to other cells will contain the track piece that connects the two neighbours in cause.
- Cells that have exactly two, certain connections to other cells will contain the track piece that connects the two neighbours in cause.
- Cells that have less than two possible connections to other cells will not contain a track.

Deductions – Repeat while puzzle changes



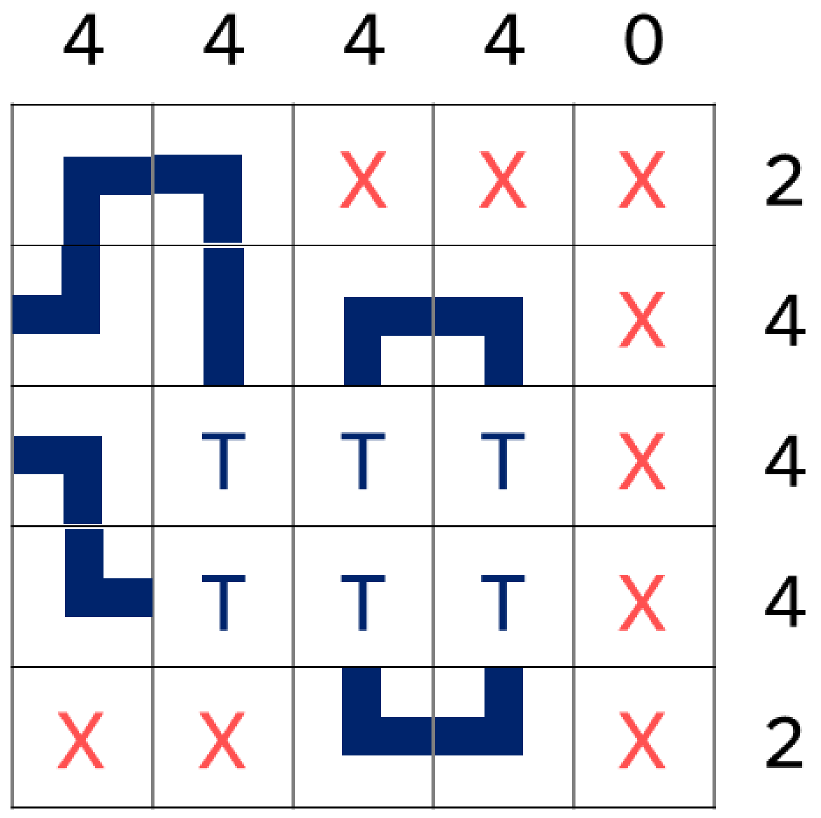
I. Implied cells

II. Enough cells

III. Enough spaces

IV. Fill tracks

Deductions – Repeat while puzzle changes



- I. Implied cells
- II. Enough cells
- III. Enough spaces
- IV. Fill tracks

Picking the next cell to assume values for

The cell with the lowest number of possible tracks*

+ Maintains the narrowest search breadth in the average case

- The definition for such a cell is loose given the uniqueness of the solution and the usefulness of the solution depends on a strategy to evaluate cells accurately and quickly

The first unknown cell on the path starting from an endpoint

+ Rigorously defined and can be accurately found in polynomial time
+ Allows for the description of several other rules

- Has very good performance on the average case, but very costly worst cases - exponential slowdown

Inference rules and constraints

I. Cells implied by a connection from another cell

II. Enough cells on a row/column

III. Enough spaces on a row/column

IV. Cells with not enough neighbors to have tracks

V. Certain cells two possible connections

VI. Cells with two known connections

VII. A path forming a bottleneck on a row/column

VIII. Remote areas (surrounded by impossible cells)

$\longrightarrow$

I. No cell can be proven to have two different values

II. No more known tracks than specified by hints

III. No more impossible cells than allowed for by the hints

IV. No mismatch in neighboring cells

V. No loops

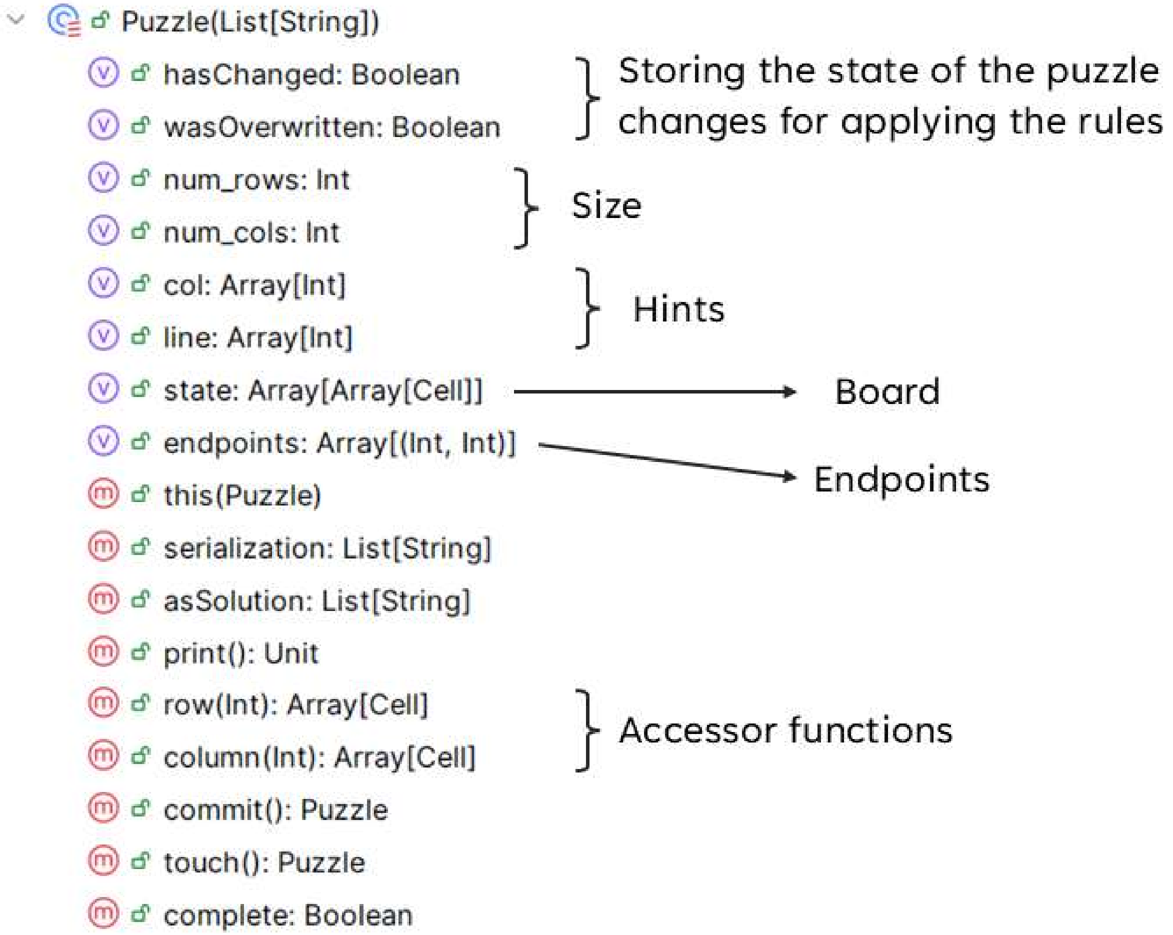
VI. No cells with more than two incoming connections

VII. No “off-by-ones” in border areas

VIII. No known cell can be isolated from the endpoints

$\xrightarrow{\text{Generate new guess}}$

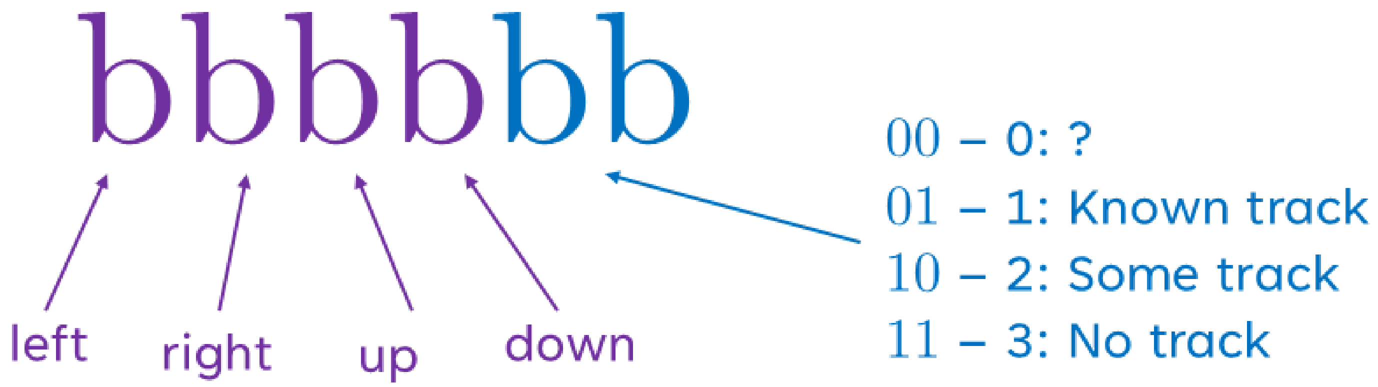
Puzzle representation



```
Puzzle(List[String])
    hasChanged: Boolean
    wasOverwritten: Boolean
    num_rows: Int
    num_cols: Int
    col: Array[Int]
    line: Array[Int]
    state: Array[Array[Cell]]
    endpoints: Array[(Int, Int)]
    this(Puzzle)
    serialization: List[String]
    asSolution: List[String]
    print(): Unit
    row(Int): Array[Cell]
    column(Int): Array[Cell]
    commit(): Puzzle
    touch(): Puzzle
    complete: Boolean
```

Storing the state of the puzzle changes for applying the rules
Size
Hints
Board
Endpoints
Accessor functions

Cell representation



00 – 0: ?
01 – 1: Known track
10 – 2: Some track
11 – 3: No track

Inference rules and constraints

III. No more impossible cells than allowed for by the hints

```
def applyTooLittleRemainingTracksConstraint(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
        )
}
```

Performance

93.5% of the tests on the evaluation test bank

maximum size of a solved puzzle: 45x45

median time spent on a puzzle: 4.048 s

Guessing order

1. ║
2. ╔
3. ╚
4. ╗
5. ╝
6. ═

Part II: Data

ACT ONE & Protocol Buffers

Part IIa: ACT-ONE

5x5 Puzzle Validation

The ACT ONE Language

- A specification description language
- Is built on top of the model of canonical term algebras

Task:

Describe a specification in the ACT ONE language to evaluate 5x5 puzzle states' validity.

Rough solution design model:

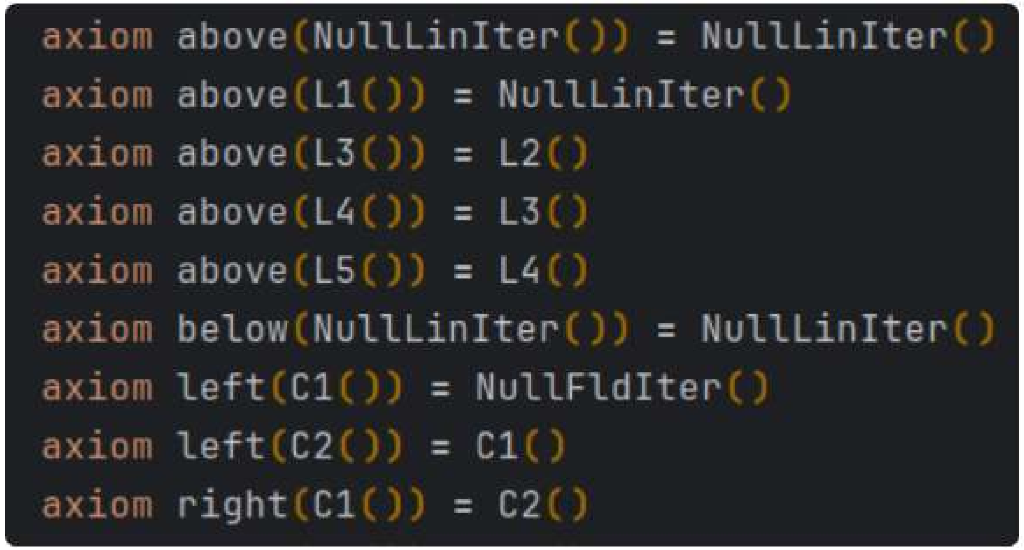
Defining a representation of a puzzle and formulating axioms and operators to reduce a representation of a 5x5 puzzle to a validity state (true/false).

Fundamental Constructors & Sorts

- Hints as atomic elements (H0 – H5)
- Fields as compositions of two directions: [West, East], [North, South], [Nowhere, Nowhere] etc
  - -> [East, East], [West, Nowhere]
- Fields as atomic elements (FEmpty, FImpossible, FCurveSouthEast, FHorizontal etc)
- Board consisting of:
  - List of 5 hints for each column
  - 5 rows consisting of 5 fields and a row hint

Utilities (observers)

- toKnownCertainPossible
- toNESWDirections
- FieldIterator & LineIterator



```
axiom above(NullLinIter()) = NullLinIter()
axiom above(L1()) = NullLinIter()
axiom above(L3()) = L2()
axiom above(L4()) = L3()
axiom above(L5()) = L4()
axiom below(NullLinIter()) = NullLinIter()
axiom left(C1()) = NullFldIter()
axiom left(C2()) = C1()
axiom right(C1()) = C2()
```

Loop detection

- Iterate over each cell
- Follow tracks lain in predefined order of directions
- Maintain step counter
    - If end reached in < 25 => None
    - Otherwise => Loop

- Drawback: Application on every cell, even those part of checked tracks

Results

100% of the valid puzzles accepted

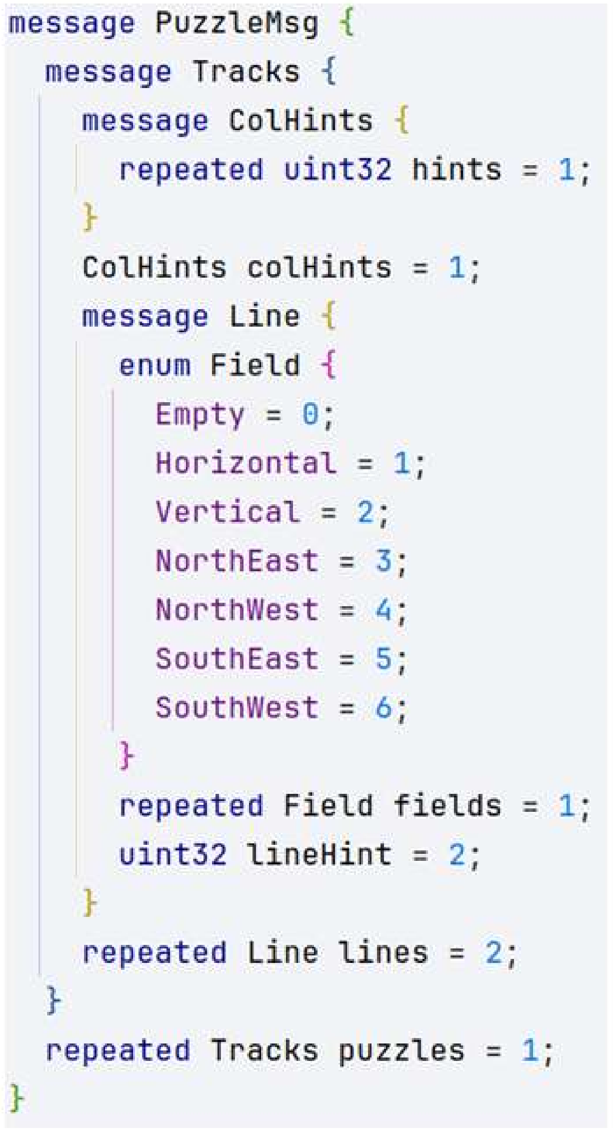
Invalid puzzles rejected: 100%

(1 invalid puzzle could not be represented)

Part IIb: Protobuf

Compact puzzle format

Puzzle Structure



```
message PuzzleMsg {
  message Tracks {
    message ColHints {
      repeated uint32 hints = 1;
    }
    ColHints colHints = 1;
    message Line {
      enum Field {
        Empty = 0;
        Horizontal = 1;
        Vertical = 2;
        NorthEast = 3;
        NorthWest = 4;
        SouthEast = 5;
        SouthWest = 6;
      }
      repeated Field fields = 1;
      uint32 lineHint = 2;
    }
    repeated Line lines = 2;
  }
  repeated Tracks puzzles = 1;
}
```

Results

100% of puzzles encoded, solved & decoded

Size reduction when using the established encoding: 42%

Part III: Prolog

Logical programming

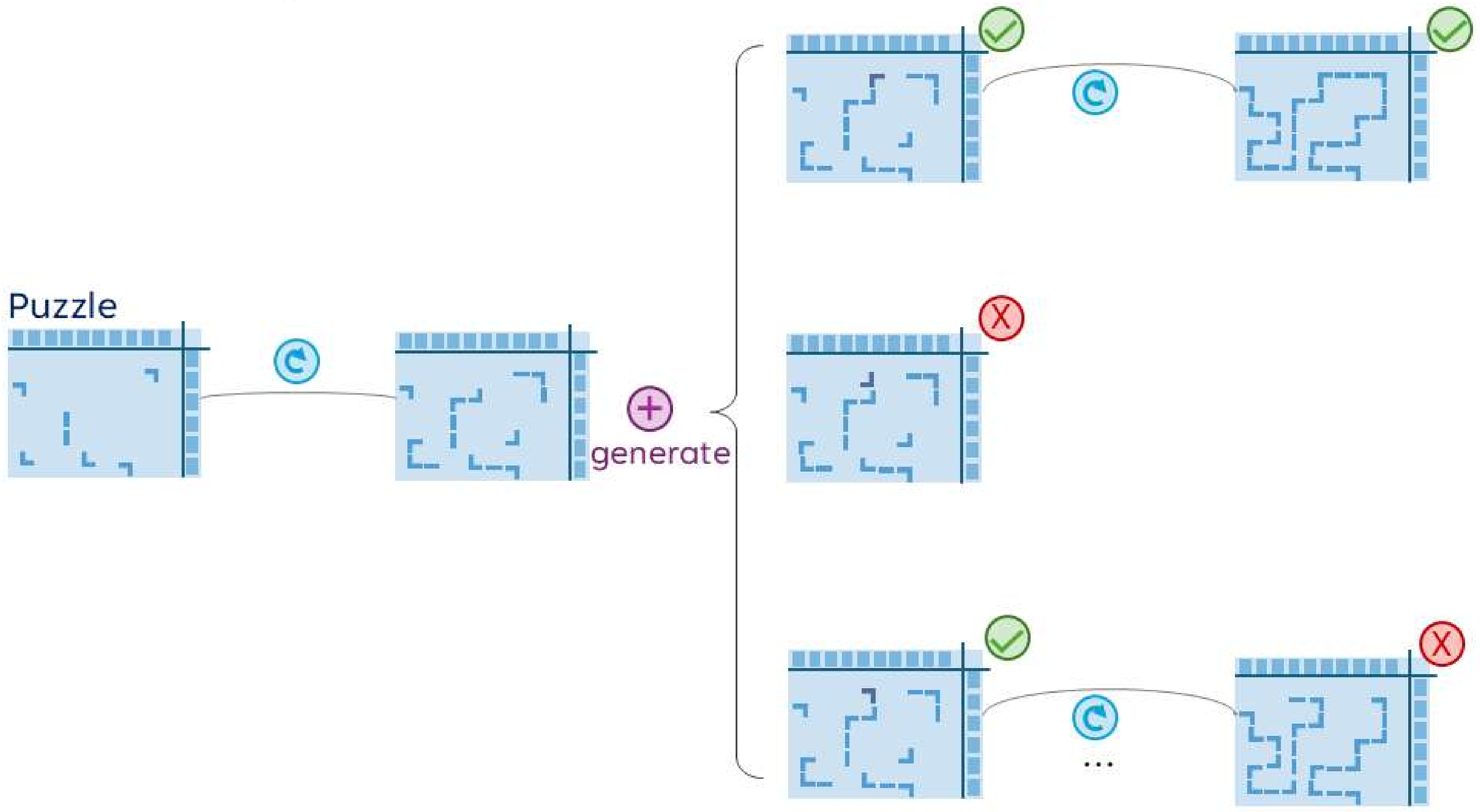
The Prolog Language

- A(n almost) purely logical, interpreted programming language
- Allows for the definition of logical predicates/clauses; the implementation employs backtracking to solve problems stated around as constraint satisfaction
- Has few para-logic features (with side-effects)

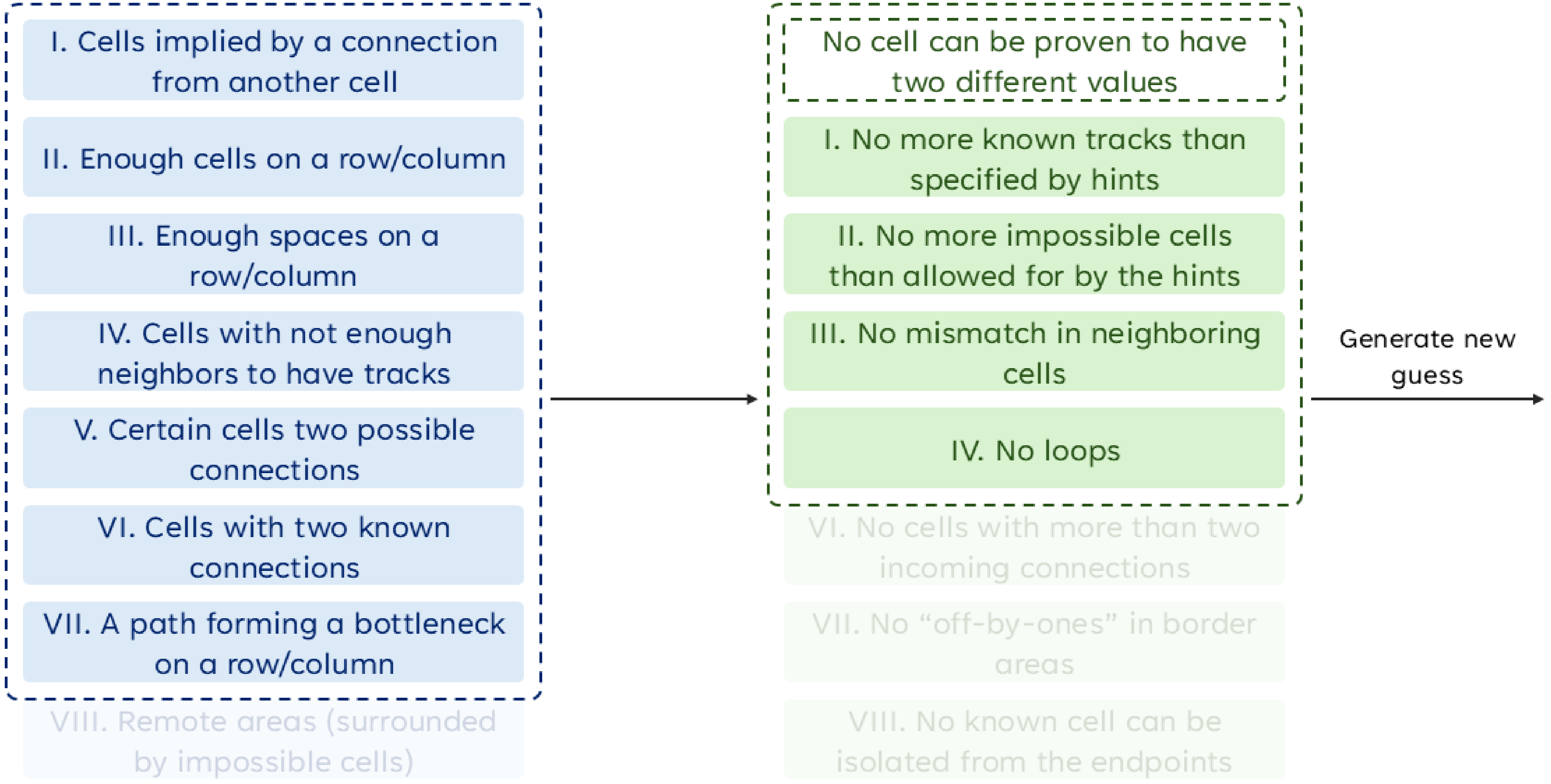
Rough solution design model:

Establishing a representation for the general case of a puzzle, defining a search space around constraints and using the Prolog language implementation to perform search in this space.

Dynamic problem model



Inference rules and constraints


I. Cells implied by a connection from another cell
II. Enough cells on a row/column
III. Enough spaces on a row/column
IV. Cells with not enough neighbors to have tracks
V. Certain cells two possible connections
VI. Cells with two known connections
VII. A path forming a bottleneck on a row/column
VIII. Remote areas (surrounded by impossible cells)
No cell can be proven to have two different values
I. No more known tracks than specified by hints
II. No more impossible cells than allowed for by the hints
III. No mismatch in neighboring cells
IV. No loops
VI. No cells with more than two incoming connections
VII. No "off-by-ones" in border areas
VIII. No known cell can be isolated from the endpoints
Generate new guess

Traversing the search space

```
solvePuzzle(T) :- applyRules(T), validate(T),!, continueSolving(T).

continueSolving(T) :- complete(T).
continueSolving(T) :- not(complete(T)), findCell(T,I,J),!, generateGuess(T,I,J),
       solvePuzzle(T).
```

Puzzle representation

```
tracks(
        5,                      } Size
        5,
        [4, 4, 4, 4, 0],          } Hints
        [2, 4, 4, 4, 2],
        [cell(1, 1, _), cell(1, 2, _), cell(1, 3, _), cell(1, 4, _),
        cell(1, 5, _), cell(2, 1, cert(nw)), cell(2, 2, cert(vert)),
        cell(2, 3, _)|...]
)                                                  → Board
```

Cell representation

```
cell(1, 2, _)

cell(1, 1, cert(_))

cell(2, 1, cert(nw))

cell(5, 1, imposs)
```

Row Column

An unknown cell can be bound to any value (impossible or any if the track pieces). An impossible cell cannot be bound to anything else, or the evaluation will fail. A certain cell can be bound to any track, but not “imposs”.

+ This representation ensures on-the-fly that the consistency of a state is maintained and will otherwise prompt a failure.

Constraint enforcing

```
checkCertOnLines(_,0).
checkCertOnLines(T,I) :- countCertLine(T, I, X), getLineHint(T, I,
X2), X=<X2,!, NextI is I-1, checkCertOnLines(T,NextI).


cellCert(cell(_,_,X)) :- not(X=imposs).
```

The checks for validity fail for certain values of the cells. Since cell states can be unknown, we do not want to perform bindings and make guesses outside the strategy we designed.
Thus, a cell is deemed certain not *if it can match the structure "cert(_)"*, but *if it cannot be imposs.*

Constraint enforcing - loops

```
visitLoop(H,W,_,cell(I,J,_),_,_,_,_) :- I=<0; J=<0; I>H; J>W.
visitLoop(H,W,_,cell(I,J,_),_,_,I,J) :- I>0, J>0, I=<H, J=<W, !, fail.

visitLoop(_,_,_,cell(I,_,cert(nw)),0,-1,_,_) :- I=<1.
visitLoop(_,_,L,cell(I,J,cert(nw)),0,-1,_,_) :- I>1, NextI is I-1,
getCell(NextI,J,L,C), not(cellKnown(C)).
visitLoop(H,W,L,cell(I,J,cert(nw)),0,-1,SI,SJ) :- I>1, NextI is I-1,
getCell(NextI,J,L,C), cellKnown(C),!, visitLoop(H,W,L,C,1,0,SI,SJ).
```

The checks for loops work by pattern matching the current cell and, according to its value, either consider the constraint satisfied, fail, or recursively defer the choice to another clause based on the next cell.
Prolog uses a more powerful pattern matcher than ACT ONE – thus, the formulation for this constraint is a bit more flexible. The start point is also given as a variable, removing the need to set the total number of cells as the maximum recursion depth.

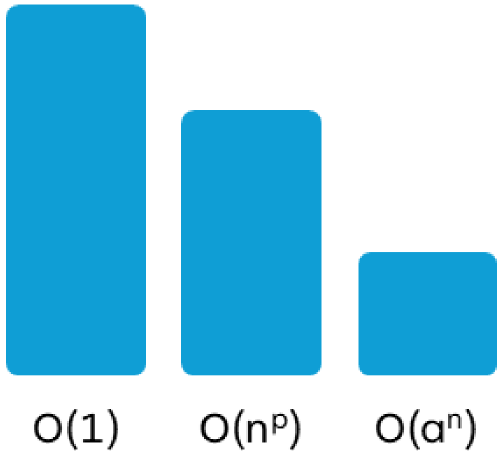
Performance

78.5% of the tests on the evaluation test bank

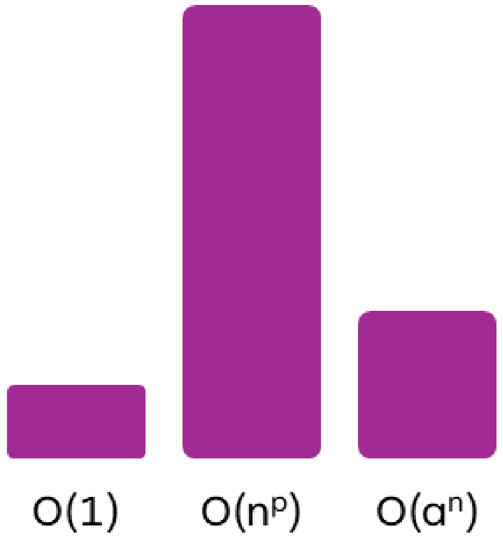
maximum size of a solved puzzle: 38x38

median time spent on a puzzle: 0.962 s

Comments on performance

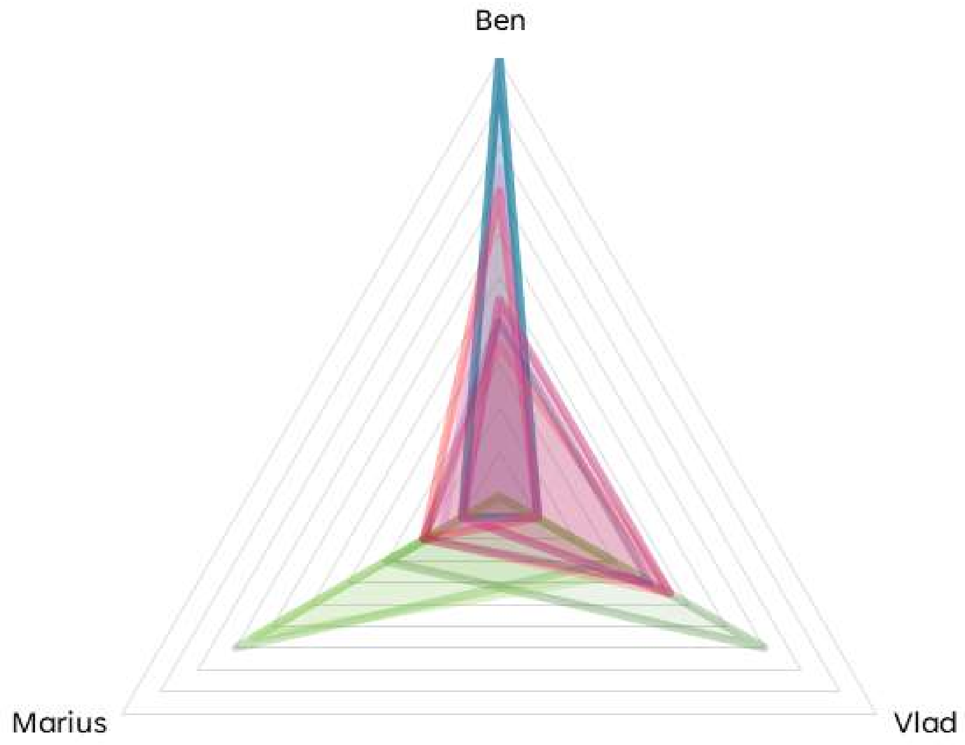


Scala



Prolog

Project management



🟦 Scala - Solution design | Ben
🟦 Scala - Solution development | Ben
🟦 Scala - Documentation | Ben
🟩 Scala - Proof of concept | V&M
🟩 Scala - Solution development | V&M
🟩 Scala - Docuemntation | V&M
🟥 Data - ACT ONE
🟥 Data - Protocol Bufffers
🟥 Data - Documentation
🟪 Prolog - Solution design
🟪 Prolog - Solution development
🟪 Prolog - Documentation

Thank you for your time!