AR part 2

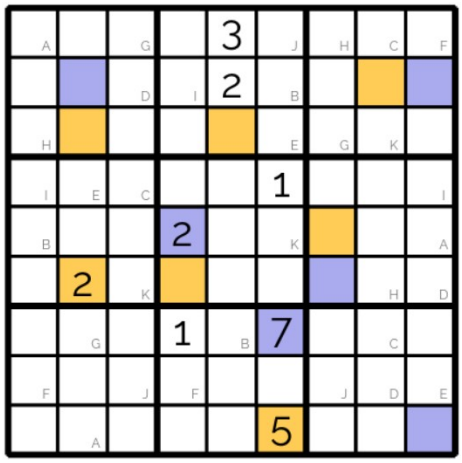
Ruben Hensen
ruben.hensen@proton.me
s1074549

October 2023

1 Special sudoku

You are the commander of a fleet of 7 ships which are lost in the mist on Sudoku Sea. Find where they are, using the following clues:

- Two of your ships have length 4, one has length 3, four have length 2.
- All the ships are free; none of them touch each other (diagonally is allowed)
- The numbers in a ship are consecutive. For example, a ship of length 4 with a 5 on one end can consist of 5-6-7-8 or 5-4-3-2.
- In the Sudoku, two neighbouring fields are only consecutive if they fall within a ship. Hence, you will never see a 3 above, below, right of or left of a 4, unless they are both in the same ship.
- Some of the digits in the Sudoku have been given, and for every ship, one field that the ship is on has been coloured yellow.
- Some of the fields that do not contain a ship have been coloured blue.



1.1 Solution

For each column $i = 0, ..., 8$ and row $j = 0, ..., 8$ we introduce two variables, an integer variable s_{ij} and a boolean variable b_{ij}. Whose meanings are defined by:

$$s_{ij} \text{ has value } x \iff \text{there is a value } x \text{ on position } (i, j)$$

$$b_{ij} \text{ is True} \iff \text{there is a ship on position } (i, j)$$

Lock in the number clues

In the sudoku, there are a few number clues. The corresponding variables can be assigned the value in the clues. Let c_{ij} equal a number clue on position (i, j). Cells without clues are ignored, in our case these clues are represented by zeros.

$$\bigwedge_{i=0}^{8} \bigwedge_{j=0}^{8} c_{ij} \neq 0 \implies s_{ij} = c_{ij}$$

Lock in the colour clues

In the sudoku, there are a few colour clues. The corresponding variables can be assigned the value in the clues. Let co_{ij} equal a colour clue on position (i, j). Cells without clues are ignored, in our case these clues are represented by 'w'.

$$\bigwedge_{i=0}^{8} \bigwedge_{j=0}^{8} (co_{ij} = \texttt{'b'} \implies \neg b_{ij}) \wedge (co_{ij} = \texttt{'y'} \implies b_{ij})$$

Every cell must have a value in the interval (1,9)

A constraint on every cell to take a value from higher than 0 or lower than 10.

$$\bigwedge_{i=0}^{8} \bigwedge_{j=0}^{8} s_{ij} \geq 1 \wedge s_{ij} \leq 9$$

Every number in a column is distinct

Every combination of two cells in a column is made. Then the constraint makes sure they are not equal. Because there are 9 cells and the values can only be from (1,9) we know for sure that all numbers are represented in every column.

$$\bigwedge_{i=0}^{8} \bigwedge_{0 \leq j1 < j2 \leq 8} s_{ij1} \neq s_{ij2}$$

Every number in a row is distinct

This constraint works the same way as the column constraint, only on rows.

$$\bigwedge_{j=0}^{8} \bigwedge_{0 \leq i1 < i2 \leq 8} s_{i1j} \neq s_{i2j}$$

Every number occurs at least once in a 3x3 block

The constraint takes a 3x3 block. In that block, a number from v should be in one of the cells. This is then repeated for all numbers (1,9) and for every block in the sudoku.

$$\bigwedge_{i2=0}^{2} \bigwedge_{j2=0}^{2} \bigwedge_{v=0}^{8} \bigvee_{i1=0}^{2} \bigvee_{j1=0}^{2} s_{(i1+i2*3)(j1+j2*3)} = v$$

Numbers are consecutive if and only if there is a ship

If there are two adjacent cells with a value of 1 apart in the number matrix, then there should be a ship represented by true booleans in the corresponding cells in the boolean matrix. This also works the other way around. If there are two adjacent true cells in the boolean matrix, the corresponding cells in the number matrix should be consecutive.

In the constraint, we go through every position (i, j). Then we create a constraint for every adjacent cell, they are consecutive if and only if there is a ship.

Deconstructing the formula, taking the "adjacent cell above constraint" as an example. $j \neq 0 \implies$ ... makes sure it is not an edge position, so that the constraint does not go out of bounds. $\left|s_{ij} - s_{i(j-1)}\right| = 1$ is the calculation to make sure the two adjacent cells are consecutive. $b_{ij} \wedge b_{i(j-1)}$ checks if the two adjacent cells are a ship. The last two formulas are then connected with an $\iff$. Lastly, the whole formula is repeated four times to check all four adjacent cells and repeated for every position (i, j).

It bears mentioning that this formula generates duplicate constraints. For example, it will generate constraints for a pair of positions $(1, 4)$ and $(1, 5)$, but also for the pair of positions $(1, 5)$ and $(1, 4)$.

$$\bigwedge_{i=0}^{8}\bigwedge_{j=0}^{8}(j \neq 0 \implies (\left|s_{ij} - s_{i(j-1)}\right| = 1 \iff b_{ij} \wedge b_{i(j-1)}))$$
$$\wedge\ (j \neq 8 \implies (\left|s_{ij} - s_{i(j+1)}\right| = 1 \iff b_{ij} \wedge b_{i(j+1)}))$$
$$\wedge\ (i \neq 0 \implies (\left|s_{ij} - s_{(i-1)j}\right| = 1 \iff b_{ij} \wedge b_{(i-1)j}))$$
$$\wedge\ (i \neq 8 \implies (\left|s_{ij} - s_{(i+1)j}\right| = 1 \iff b_{ij} \wedge b_{(i+1)j}))$$

If there is a ship, there should not be any adjacent ships.

Making sure the boats are not adjacent to each other works fairly similarly to the previous constraint. In this constraint, all cells are gone through one-by-one, and for every cell a constraint is made for its adjacent cells.

The constraint is fairly easily explained with a figure. In figure 1 we can see the constraint worked out for the adjacent cell on the right of the current cell. If the cell and its adjacent cell are true, then the four cells above and below (or left and right if we check a vertical ship) cannot be a ship and are thus false. We repeat this for every adjacent block, and we repeat it once more for every block in the sudoku.

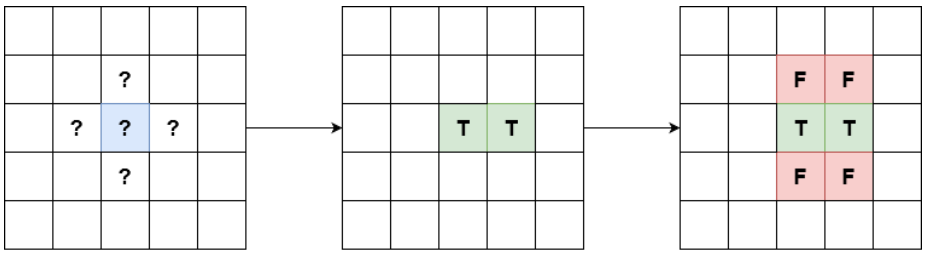


Figure 1: A single cell constraint to make sure boats are not adjacent to each other

Going through the formula piece by piece. $\bigwedge_{i=0}^{8}\bigwedge_{j=0}^{8}$ repeats the constraint for every cell. $(j \neq 0 \implies (i \neq 0 \implies$ ... makes sure the following constraint does not go out of bounds. $b_{ij} \wedge b_{i(j-1)} \implies$ ... checks if the current and adjacent square are true. $\neg b_{(i-1)j} \wedge \neg b_{(i-1)(j-1)})$ if it is true, make sure the correct two adjacent cells are false (we can only do 2 of the 4 false cells, because we need to check for bounds). This is then repeated for all four adjacent cells.

$$
\begin{aligned}
\bigwedge_{i=0}^{8}\bigwedge_{j=0}^{8}(j \neq 0 \implies (i \neq 0 \implies (\ & b_{ij} \ \wedge\ b_{i(j-1)} \implies \neg b_{(i-1)j} \ \wedge\ \neg b_{(i-1)(j-1)}) \\
& \wedge (\ b_{ij} \ \wedge\ b_{(i-1)j} \implies \neg b_{i(j-1)} \ \wedge\ \neg b_{(i-1)(j-1)}))) \\
\wedge (j \neq 0 \implies (i \neq 8 \implies (\ & b_{ij} \ \wedge\ b_{i(j-1)} \implies \neg b_{(i+1)j} \ \wedge\ \neg b_{(i+1)(j-1)}) \\
& \wedge (\ b_{ij} \ \wedge\ b_{(i+1)j} \implies \neg b_{i(j-1)} \ \wedge\ \neg b_{(i+1)(j-1)}))) \\
\wedge (j \neq 8 \implies (i \neq 0 \implies (\ & b_{ij} \ \wedge\ b_{i(j+1)} \implies \neg b_{(i-1)j} \ \wedge\ \neg b_{(i-1)(j+1)}) \\
& \wedge (\ b_{ij} \ \wedge\ b_{(i-1)j} \implies \neg b_{i(j+1)} \ \wedge\ \neg b_{(i-1)(j+1)}))) \\
\wedge (j \neq 8 \implies (i \neq 8 \implies (\ & b_{ij} \ \wedge\ b_{i(j+1)} \implies \neg b_{(i+1)j} \ \wedge\ \neg b_{(i+1)(j+1)}) \\
& \wedge (\ b_{ij} \ \wedge\ b_{(i+1)j} \implies \neg b_{i(j+1)} \ \wedge\ \neg b_{(i+1)(j+1)})))
\end{aligned}
$$

Count the number of ships with length 2-8

We can only have a certain number of ships of a certain length. To check this, we use a counting constraint. We check for True variables in a row, the same length of the ship we are counting, with a false variable on either side. In figure 2 is an example of the constraints created for a single row. If any of these constraints are true, the counter of ships with length 5 in increased by 1. We do this for every row and every column. Lastly, we sum all the constraint and make sure it is equal to the amount of ships specified in the problem.

The formula is split in multiple parts. To make sure we do not go out of bounds we have three parts for rows (left, middle and right). And three parts for columns (up, middle and down). In figure 2 the first row is an example of a left constraint and the last row is an example of right constraint. The other two are examples of middle constraints. All these constraints are generated for every ship length $l = 2, 3, ..., 8$. For ships of length one we have a separate constraint.

Lastly, let m_l equal the maximal number of ships allowed of length $l/$

$$
\begin{aligned}
&\sum_{l=1}^{7}(\\
&\sum_{j=0}^{8}\mathrm{If}((\bigwedge_{i=0}^{l-1} b_{ij}) \ \wedge \ \neg b_{lj}, 1, 0)+ \\
\sum_{k=0}^{8-l}\sum_{j=0}^{8} l \neq 8 \implies &\mathrm{If}(\neg b_{(0+k)j} \ \wedge \ (\bigwedge_{i=0}^{l-1} b_{(i+k+1)j}) \ \wedge \ \neg b_{(l+k+1)j}, 1, 0)+ \\
&\sum_{j=0}^{8}\mathrm{If}((\bigwedge_{i=0}^{l-1} b_{ij}) \ \wedge \ \neg b_{(8-l)j}, 1, 0)+ \\
&\sum_{j=0}^{8}\mathrm{If}((\bigwedge_{i=0}^{l-1} b_{ji}) \ \wedge \ \neg b_{jl}, 1, 0)+ \\
\sum_{k=0}^{8-l}\sum_{j=0}^{8} l \neq 8 \implies &\mathrm{If}(\neg b_{j(0+k)} \ \wedge \ (\bigwedge_{i=0}^{l-1} b_{j(i+k+1)}) \ \wedge \ \neg b_{j(l+k+1)}, 1, 0)+ \\
&\sum_{j=0}^{8}\mathrm{If}((\bigwedge_{i=0}^{l-1} b_{ji}) \ \wedge \ \neg b_{j(8-l)}, 1, 0)) = m_l
\end{aligned}
$$

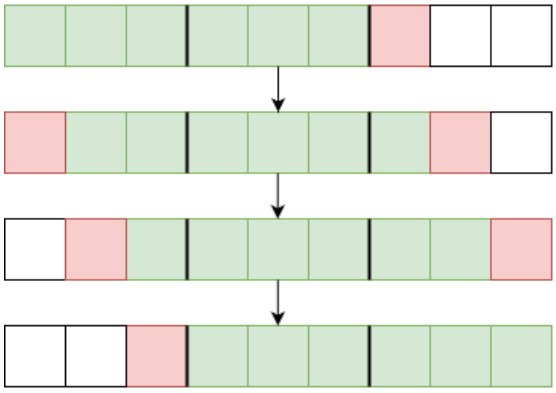


Figure 2: Example of counting ships of length 6 in a single row.

Count the number of ships with length 9

$$
\sum_{j=0}^{8}\mathrm{If}(\bigwedge_{i=0}^{8} b_{ij}, 1, 0) + \sum_{i=0}^{8}\mathrm{If}(\bigwedge_{j=0}^{8} b_{ij}, 1, 0) = m_9
$$

No ships of length 1

I made the assumption that ships of length 1 are not allowed, as they are no ships of size one in Battleship. A version that supports would be possible, however, it would create considerable rewriting of my program which I do not have time for.

To make sure there are no ships of length one, a simple constraint is made on every cell of the boolean matrix. If the cell is true, then all its adjacent cells cannot be false all at once. Or in different words, there must be at least one adjacent cell that is also true.

$$\bigwedge_{i=0}^{8}\bigwedge_{j=0}^{8} b_{ij} \implies \neg(i \neq 0 \implies \neg b_{(i-1)j} \wedge$$
$$i \neq 8 \implies \neg b_{(i+1)j} \wedge$$
$$j \neq 0 \implies \neg b_{i(j-1)} \wedge$$
$$j \neq i \implies \neg b_{i(j+1)})$$

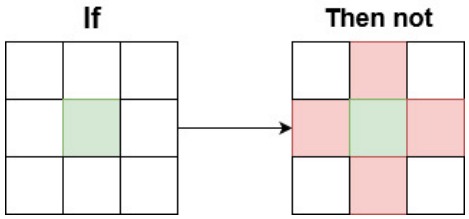


Figure 3: Constraints to prevent boats of length one

1.2 Results

Solve the Sudoku

```
Solution found.
2  9  5 |7  3* 6 |1  8* 4
4  6  1 |5  2* 8 |3  7* 9
7* 8* 3 |9  1* 4 |6  2  5
--------+--------+--------
3  5  7 |4  8  1 |9* 6  2
6  1* 9 |2  5  3 |8* 4  7
8  2* 4 |6* 7* 9 |5  1  3
--------+--------+--------
5  3* 8 |1  4  7 |2  9  6
1  4* 6 |3  9  2 |7  5  8
9  7  2 |8  6* 5*|4* 3* 1
```

Write a general program

Check.

2 Seminar groups

In the TIME seminar at Science Fiction University, all students following the class have to read a different book on time travel, and give a presentation about what they learned. Each book is supervised by a professor, who is an expert on the topics discussed in that book. Theassignment of books to students works as follows:

- Each of the professors for the course selects 1–3 books.
- The students are presented with a list of books to choose from. They make a preference, ranking all books in the order they want to read them (ranking two books equally is allowed).
- The best possible assignment of books to students is chosen, taking preferences into account, but not giving two students the same book. Here, assignment A is better than assignment B if the same number of students have their worst preference, second-worst preference and so on, until for choice n, fewer students in A get choice n than in B.

For example: if there are two professors, with books A1, A2 (for professor A) and books B1, B2, B3 (for professor B), students might give the following preference:

- Alice has preference: A1 : 1, A2 : 4, B1 : 2, B2 : 2, B3 : 4 (so her first choice is A1, her shared second choices are B1 and B2, and her shared fourth choices are A2 and B3).
- Bob has preferences A1 : 1, A2 : 2, B1 : 3, B2 : 5, B3 : 4.
- Carol has preferences A1 : 4, A2 : 2, B1 : 1, b2 : 5, B3 : 2
- Dennis has preferences A1 : 1, A2 : 2, B1 : 5, B2 : 3, B3 : 3

Then (Alice: A1, Bob: A2, Carol: B1, Dennis: B2) is better than (Alice: B1, Bob: A1, Carol: A2, Dennis: B2) because in both assignments one person gets their third choice (Dennis), but in the first assignment only one person gets their second choice (Bob), while in the second two people get their second choice (Alice and Carol). The best possible assignment is (Alice:B2, Bob:A1, Carol:B1, Dennis: A2), since then no one gets their third choice, and only two people get their second choice. Of course, in a real class, the problem is a bit larger and harder to oversee!

(a) Find the best possible assignment of students to books for the preference assignment given in seminar.csv in the assignment documents.

(a) Find the best possible assignment where no professor has to supervise more than 2 books! (For example, A1, A2 and A3 cannot all be assigned.)

(a) Write a program that reads the student preferences and returns a best-possible assignment, taking into account that no professor wants to supervise more than 2 books. A template Python file is provided.

Tip:. you can call the SMT solver more than once. (With different input.)

2.1 Solution

We generalize the problem by making the number of students, professors and books variable, we name the variables for these groups s_n, p_n and b_n respectively. For students $s = 0, 1, ...s_n - 1$, professors $p = 0, 1, ..., p_n - 1$ and books $b = 0, 1, ..., b_n - 1$ we create a boolean variable for every combination of s, p and b called X_{spb} . X can be seen as a three-dimensional matrix containing all the possible combinations. In this matrix, a true variable represents a book assigned to a specific student and professor. We add multiple constraints to X such that the books can only be divided as intended by the rules. Constraints such as 1 book per student, and every professor can only supervise the books he or she is specialized in. In this state we can use the sat solver to create a solution in which every one gets a book. This does not yet take into account their preference.

To implement the preference, we go through the problem in a loop based on the preference number, starting with the lowest preference (the highest preference number) and counting up. Let us call the current preference number f. We try to minimize the sum of student/book combinations that have preference f . We do this by setting a constraint of 0 on the sum of student/book combinations that equal f. If this problem is satisfiable, it means we can solve the problem without anybody having a book with preference f. Thus we can keep this constraint and move on to the next preference number and try again. If the problem is not satisfiable, we increase the maximum sum by 1. We keep doing this until we find a solution for preference f. Then we continue to the next preference.

By minimizing each preference sum starting from the lowest preference, we make sure it is not possible to find any combination with better preferences.

An example run could look like this:

- We try to solve the problem with 0 students having a book of preference 20. This is satisfiable, so we leave this constraint and move on to preference 19.
- We try to solve the problem with 0 students having a book of preference 19. This is also satisfiable, so we leave this constraint and move on to preference 18.
- We try to solve the problem with 0 students having a book of preference 18. This is unsatisfiable. So we remove this constraint, and try again to solve with 1 student having a preference of 18.

- We try to solve the problem with 1 students having a book of preference 18. This is unsatisfiable. So we remove this constraint, and try again to solve with 1 student having a preference of 18.
- We try to solve the problem with 2 students having a book of preference 18. This is satisfiable, so we leave the constraint and move on to preference 17.
- We try to solve the problem with 0 students having a book of preference 17. ...
- ...
- We try to solve the problem with 9 students having a book of preference 1. This is satisfiable. There are not any higher preferences, thus we end the program and print the result.

Every student is assigned 1 book

We sum the matrix in the professor and books dimension. This way we get an array of the amount of book/professor combinations a student is assigned. This number should be one. We count the booleans with the If function as specified by Z3. If the boolean is true, we turn it into a 1, otherwise we fill in 0. This way we can sum different things in the matrix.

$$\bigwedge_{i=0}^{s_n-1} \left(\sum_{j=0}^{p_n-1} \sum_{k=0}^{b_n-1} \text{If}(X_{ijk}, 1, 0)\right) = 1$$

Every book has at most 1 student

If every student is assigned one book, a single book can still be assigned to two students. We fix this by introducing a constraint that a book can only have 1 or 0 students. We achieve this in the same way as the previous constraint, only we sum in the student and professor dimension this time.

$$\bigwedge_{k=0}^{b_n-1} \left(\sum_{i=0}^{s_n-1} \sum_{j=0}^{p_n-1} \texttt{If}(X_{ijk}, 1, 0)\right) \leq 1$$

Every professor supervises at most 2 books

This constraint is optional. We run the solver once with, and once without, this constraint.

$$\bigwedge_{j=0}^{p_n-1} \left(\sum_{i=0}^{s_n-1} \sum_{k=0}^{b_n-1} \texttt{If}(X_{ijk}, 1, 0)\right) \leq 2$$

Every professor only supervises the books they specialize in

The matrix is a Cartesian product of students, professors, and all books. Therefore, there are many incorrect book/professor combinations in which the book is not a book the professor specializes in. We need to filter out all these wrong combinations.

The formula uses takes all the combinations. If book number k is not in the set of books the professor number j specializes in. To get this set, we use the `books_for()` function. We make all the combination in the set false.

In the formulation, `books_for()` takes an integer representing a professor, then returns a set of integers corresponding to the books that professor specializes in.

The code version is slightly different. It takes the name (letter) of a professor, and returns a list of book names (codes).

$$\bigwedge_{0 \leq i \leq s_n, 0 \leq j \leq p_n, 0 \leq k \leq b_n \ \wedge \ k \notin \texttt{books\_for}(j)} \neg X_{ijk}$$

Maximal n number of students with preference f

This is the constraint used to find the correct division of books. We reuse this same constraint multiple times with different values.

The formula is quite simple. It filters all the combinations of students and books that have the preference f. It uses the `If` function trick to make the booleans summable. Then it sums up all these variables and makes sure they are below or equal to the max number of student n.

$$\left(\sum_{0 \leq i \leq s_n, 0 \leq j \leq p_n, 0 \leq k \leq b_n \ \wedge \ \text{rank}(i,k) = f} \texttt{If}(X_{ijk}, 1, 0) \right) \leq n$$

2.2 Results

Professor without max number of books:

```
Alice: A1 (2)
Bob: B1 (1)
Carol: F2 (3)
Dennis: D2 (1)
Eliza: E1 (1)
Fred: G1 (2)
Georgianna: H2 (3)
Harry: J3 (1)
Ilse: C2 (2)
John: F1 (1)
Karen: C3 (1)
Leo: J1 (2)
Monica: C1 (1)
```

```
Nico: G3 (3)
Olivia: J2 (1)
Patrick: H1 (3)
```

Professor with at most 2 books:

```
Alice: A1 (2)
Bob: B1 (1)
Carol: F2 (3)
Dennis: D2 (1)
Eliza: E1 (1)
Fred: G1 (2)
Georgianna: C3 (1)
Harry: I1 (4)
Ilse: C2 (2)
John: F1 (1)
Karen: I2 (4)
Leo: J1 (2)
Monica: H2 (1)
Nico: G3 (3)
Olivia: J2 (1)
Patrick: H1 (3)
```

3 Robot planning

A robot starts in one of the green cells with a star. The goal is to lead the robot to one of the black cells marked with a star in as few steps as possible. That is, we want to limit the length of the trajectory from start to target in the worst-case (over all starting positions) by a number X. It suffices to solve the decision problem: Given a value for X, is there a plan so that we will reach a target within X steps from every start cell? In particular, plans are mappings from every cell to one of the possible directions N, S, W, E. Valid plans adhere to the following rules:

- A robot can move in one of the four cardinal directions: up (or N), down (or S), left (or W), right (or E). It will then reach the adjacent cell. It is allowed to stand still on an end point.
- Robots cannot fall off the grid. Instead, if they move towards a boundary, they will just stay where they are.
- The robot does not know where it is, but it can detect the surface type. Therefore, in every cell with the same surface type (described by a number/color), the robot must move in the same direction. The robot will not remember previous terrain types.
- A robot will catch fire if it visits a red lava cell (with a black cross). This makes it impossible to reach the end.

- On sticky patches (white with blue diamonds), the robot needs 7 time steps to actually finish the movement.

Write a program that solves the decision problem.

3.1 Solution

We generalize the problem as follows. We have four sets, A, O, X, Y for all actions, observations, size of the x dimension and size of the y dimension respectfully. In these sets all elements are numbered starting from zero and increasing by one. For example, actions would be $A = 0, 1, 2, 3$. Let a set with subscript n mean the maximum number in that set, e.g. $A_n = 3$. We create set of boolean variables with all combinations of actions and observations, we call this set P. P can be seen as a two-dimensional matrix, in which every element can be reached by $A_{o,a}$. We also create a reachability matrix R, filled with all $X \times Y$. Individual elements are represented by $R_{x,y}$. The third matrix is a time matrix T of $X \times Y$, in which every element is reachable with $T_{x,y}$.

Lastly, we define $Start$, which contains tuples of all the starting coordinates. $succ(x, y, a)$ which takes a coordinate and an action, then returns the successor tile. $pred(x, y, a)$ which takes a coordinate and an action, then returns the predecessor tile. And $o_{(x,y)}$, which represents the observation number of tile (x, y).

With all the correct constraints in place, we create a final constraint limiting the maximal amount of steps path may be. Then we loop over this constraint, increasing it by one every time, until a solution is found.

Every observation has 1 action

We make sure every observation has exactly 1 action associated with it. Except for the lava tile (nr. 2), which we give 0 actions.

$$\bigwedge_{o=0}^{1} \left(\sum_{a=0}^{A_n} \text{If}(P_{o,a}, 1, 0)\right) = 1$$

$$\bigwedge_{o=3}^{O_n} \left(\sum_{a=0}^{A_n} \text{If}(P_{o,a}, 1, 0)\right) = 1$$

$$\bigwedge_{a=0}^{A_n} \neg P_{2,a}$$

Starting cells are reachable

$$\bigwedge_{(x,y) \in Start} R_{x,y}$$

If the cell is reachable, its predecessor is also reachable and has an action towards the current cell

$$\bigwedge_{(x,y)\in R} R_{(x,y)} \implies \bigvee_{(x',y')} \wedge A_{o(x',y'),a}$$

where $(x', y') = pred(x, y, a)$, and, $o(x', y')$ is the observation in (x', y'). We create this constraint for all cells, in all cardinal directions. If the *pred* function returns an out-of-bounds cell, no constraint is created for that cardinal direction.

If the cell is reachable and has an action towards some next cell x, then cell x is also reachable.

$$\bigwedge_{(x,y)\in R} \bigwedge_{a\in A} R_{(x,y)} \wedge A_{o(x,y),a} \implies R_{(x',y')}$$

where $(x', y') = succ(x, y, a)$. We create this constraint for all cells, in all cardinal directions. If the *succ* functions reaches an out-of-bounds cell. We set the implied statement to False. This is to prevent paths going of the board. The rules state that a robot would stay on the same square if it tries to run out-of-bounds. However, this would mean that the robot would get stuck on a single square and never reach its destination. Therefore, we set these constraints to False, to prevent any of these paths being allowed.

Every integer in T must be between 0 and *maxsteps*

In this constraint the *maxsteps* variable starts at 0 and is constantly increased by 1 if no solution is found. This is to find the minimum amount of steps possible.

$$\bigwedge_{(x,y)\in T} T_{x,y} \geq 0 \wedge T_{x,y} \leq maxsteps$$

Every starting cell starts with time (minimum) 1

$$\bigwedge_{(x,y)\in Start} T_{x,y} \geq 1$$

The arrival time of a successor must be greater

$$\bigwedge_{(x,y)} \bigwedge_{a\in A} \wedge P_{o_{(x,y)},a} \implies T_{x',y'} \geq T_{x,y}$$

We generate constraints for all cells in all cardinal directions, with two exceptions. The current cell is a finishing cell. Finishing cells are excluded from counting because we start counting at 1. Also, the successor cell must not be out of bounds.

The arrival time of a sticky successor must be at least +7 greater

$$\bigwedge_{(x,y)} \bigwedge_{a \in A} \wedge P_{o_{(x,y)},a} \implies T_{x',y'} \geq T_{x,y} + 6$$

The same restrictions apply as the previous constraint, with one extra. The successor state must be sticky.

3.2 Results

Grid	0	1	2	3	4	5	6	7	8	9
Min steps	15	14	31	17	8	14	20	12	18	28

Grid	10	11	12	13	14	15	16	17	18	19
Min steps	24	19	14	21	21	17	17	9	21	12

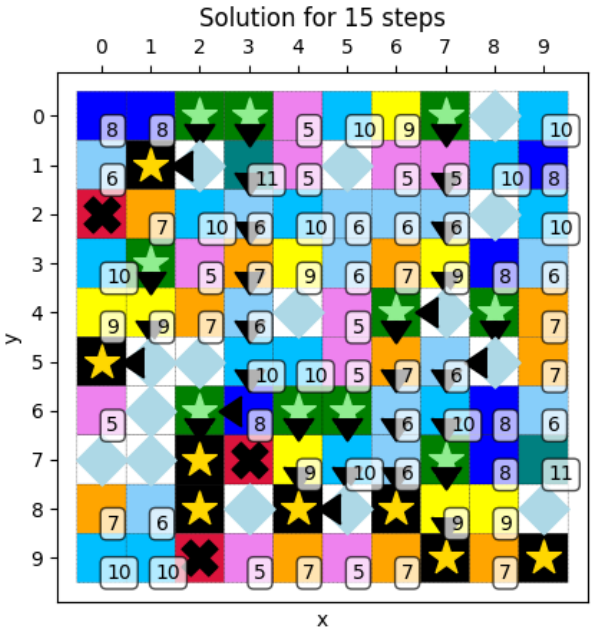


Figure 4: Solution to grid 0

4 The Multiset Path Ordering

There are many extensions and variations of the recursive path ordering we have discussed in the lecture. We will here discuss the Multiset Path Ordering

(MPO). This is an ordering on terms that is generated from a total quasi-ordering relation $\trianglerighteq$ on function symbols, using multiset comparison. Specifically, we are given a relation $\trianglerighteq$ on function symbols such that for all symbols a, g, we have: $a \trianglerighteq g$ or $g \trianglerighteq a$. Here, $\trianglerighteq$ is transitive (if $a \trianglerighteq g \trianglerighteq h$ then $a \trianglerighteq h$) and reflexive (always $a \trianglerighteq a$). We denote $a \triangleright g$ if $a \trianglerighteq g$ and not $g \trianglerighteq a$. We denote $a \equiv g$ if $a \trianglerighteq g$ and $g \trianglerighteq a$. The relations $\succsim$, $\succ$ and $\approx$ on terms are given by the following inductive definition:

1. $s \succsim t$ if and only if $s \succ t$ or $s \approx t$
2. $s \succ t$ if and only if $s = a(s_1, ..., s_n)$ and at least one of the following holds:
   (a) $s_i \succsim t$ for some $i \in \{ 1, ..., n \}$
   (b) $t = g(t_1, ..., t_m)$ with $a \triangleright g$ and $s \succ t_i$ for all $i \in \{ 1, ..., m \}$
   (c) $t = g(t_1, ..., t_m)$ with $a \equiv g$ and $\{ \{ s_1, ..., s_n \} \} \succ_{mul} \{ \{ t1, ..., tm \} \}$
3. $s \approx t$ if and only if one of the following holds:
   (a) s and t are both the same variable;
   (b) $s = a(s_1, ..., sn)$ and $t = g(t1, ..., t_n)$ and $a \equiv g$ and $\{ \{ s1, ..., sn \} \} \approx_m atch \{ \{ t1, ..., tn \} \}$

Here, a multiset is a set that may contain repetitions; order does not matter, but count does. For example, $\{ \{ 1, 1, 2, 3 \} \} = \{ \{ 1, 3, 2, 1 \} \}$ but this is not the same as $\{ \{ 1, 2, 3, 3 \} \}$. For two multisets A, B we say that $A \approx_{match} B$ if and only if we can write $A = \{ \{ s_1, ..., s_n \} \}$ and $B = \{ \{ t_1, ..., t_n \} \}$ and each $s_i \approx t_i$ (note that we can order the elements of A and B in anyway!). We say that $A \succ_{mul} B$ if we can write $A = A_1 \bigcup A2$, $B = B1 \bigcup B2$, $A2$ is non-empty, $A1 \approx_{match} B1$ and for all $b \in B2$ there is $a \in A2$ such that $a \succ b$. For example, if we take for $\succ$ the usual order on N then we have $A = \{ \{ 5, 3, 1, 1 \} \} \succ_{mul} \{ \{ 4, 3, 3, 1 \} \} = B$, by choosing $A1 = B1 = \{ \{ 1, 3 \} \}, A2 = \{ \{ 1, 5 \} \} and B2 = \{ \{ 3, 4 \} \}$. See the tip below for how to encode the multiset order by an SMT formula.

1. Find a symbol ordering such that the following inequalities all simultaneously hold, where x, y, z, u, v are all variables, and all other symbols are functions.
   - $f(a(x, y), g(x, y), a(x, g(z, u))) \succ f(a(x, z), y, g(g(g(y, x), x), x))$
   - $c(x, y, u, v) \succ f(a(x, y), f(u, u, u), g(v, f(x, y, u)))$
   - $h(g(x, g(u, z)), c(x, y, x, z)) \succ a(d(x, z), u)$
   - $a(f(x, y, z), u) \succ h(u, a(x, h(y, x)))$
   - $h(d(a(x, y), g(u, v)), a(x, y)) \succ a(c(u, x, v, y), g(y, x))$
   - $f(b(x, y, z), y, x) \succ c(x, x, y, x)$

   Provide the symbol ordering, and give the (human-readable) proof for the third inequality.

2. Can the inequalities be oriented with any other symbol ordering?

3. Write a program that, given a term rewriting system, prints YES along with the precedence if it can be ordered using MPO, and that prints NO if it cannot be ordered using MPO.

4.1 Solution

We create an integer variable for all function symbols, we put these in a set P where every element is reachable by P_f in which f represents a function symbol. For every rule, we create a cartesian product of all subterms u of s and all subterms v of t (in which s is the left-hand side of the formula and t is the right-hand side), and generate variables representing $[u > v]$ and $[u \geq v]$. We also generate a variable $[s > t]$ and $[s \geq t]$. Then we start generating constraints. Let all variables be contained in a set X in which every variable can be reached by $X_{l>r}$ or $X_{l\geq r}$ in which l represents the left-hand side and r the right-hand side.

For all rules, $[s > t]$

A trivial constraint. Let R contain all rules in tuple form (l, r).

$$\bigwedge_{(s,t)\in R} X_{s>t}$$

Sub, copy and mul constraints

For a single rule, every pair of subterms is iterated through. Based on the shape of the subterm constraints are created. Let u represent the subterm on the left hand-side and v the subterm on the right hand-side. For every subterm only one of the constraints is generated, whichever is matched first going from top to bottom:

- u **is a variable.** Variables are not greater than anything. Therefore, we create a not constraint.
$$\bigwedge_{x\in X} \neg x$$

- u **is a function application and** v **is a variable.** We create a constraint implementing part of the sub rule. Checking if any argument in u is bigger or equal than v. Let $argsu$ represent all the arguments in u.
$$\bigvee_{arg\in argsu} X_{arg\geq v}$$

- u **is a function application,** v **is a function application. The functions symbols are the same.** We create a constraint implementing the

combination of the sub rule and the multiset rule. The sub rule constraint is the same as before.

$$orExprs = \bigvee_{arg \in argsu} X_{arg \geq v}$$

The second part implements the multiset constraint. To create the multiset constraint we introduce two new types of variables. For every argument in u we create a boolean variable eq. We put these variables in a set EQ in which every element is reachable with EQ_n. Then for every argument v we create an integer variable φ. We put these variables in a set Φ in which every element is reachable with Φ_m. These variables are to encode the EQ and φ functions as described in the exercise: For ordered sequences $s_1, ..., s_n$ and $t_1, ..., t_m$, we have $\{\{ s_1, ..., s_n \}\} >_{mul} \{\{ t_1, ..., t_m \}\}$ if there exist functions $EQ : \{ 1, ..., n \} \to \{ \top, \bot \}$ and $\varphi : \{ 1, ..., m \} \to \{ 1, ..., n \}$ such that:

- for all $i \in \{ 1, ..., m \}$:
  - if $EQ(\varphi(i)) = \bot$ then $s_{\varphi(i)} \succ t_i$
  - if $EQ(\varphi(i)) = \top$ then $s_{\varphi(i)} \approx t_i$
  - if $EQ(\varphi(i)) = \top$ then i is the only index mapping $\varphi(i)$; that is, there is no $i' \in \{ 1, ..., m \} \setminus \{ i \}$ with $\varphi(i) = \varphi(i')$;
- there is at least one $j \in \{ 1, ..., n \}$ with $EQ(j) = \bot$

That is, $EQ(j)$ indicates that $s_j \in A_1$. We have $\{\{ s_1, ..., s_n \}\} \approx_{match} \{\{ t_1, ..., t_m \}\}$ if $n = m$ anmd there exist a function $\varphi : \{ 1, ..., m \} \to \{ 1, ..., n \}$ such that for all $i \in \{ 1, ..., m \}$:

- $s_{\varphi(i)} \approx t_i$
- for all $i' \in \{ 1, ..., m \} \setminus i : \varphi(i') \neq \varphi(i)$

To implement these functions we place constraints on the just created variables.

If EQ returns false on m, then the n^{th} argument of u should be bigger than the m^{th} argument of v. Let $argsu_n$ and $argsv_m$ be the n^{th} and m^{th} argument of $argsu$ en $argsv$ respectively.

$$eqTrue = \bigwedge_{n=0}^{|argsu|} \bigwedge_{m=0}^{|argsv|} \Phi_m == n \ \wedge \ \neg EQ_n \implies X_{argsu_n > argsv_m}$$

If EQ returns true on m, then the n^{th} argument of u should be bigger than the m^{th} argument of v.

$$eqFalse = \bigwedge_{n=0}^{|argsu|} \bigwedge_{m=0}^{|argsv|} \Phi_m == n \ \wedge \ EQ_n \implies X_{argsu_n \approx argsv_m}$$

At least one j returns bottom

$$justOne = (\sum_{n=0}^{|argsu|} \texttt{If}(\neg EQ_n, 1, 0)) = 1$$

Lastly, we add all the constraints together, combined with the partial sub rule, to create one single constraint.

$$X_{u>v} \implies orExprs \vee (eqTrue \wedge eqFalse \wedge justOne)$$

- u **is a function application,** v **is a function application. The functions symbols are the different.** Just as before we create a constraint implementing the combination of the sub rule and the multiset rule. The sub rule constraint is the same as before.

$$orExprs = \bigvee_{arg \in argsu} X_{arg \geq v}$$

For the other part of the constraint, we implement the copy rule. It is the exact same way the copy rule is implemented in the lpo example program. Only is the symbol order check changed from a $>$ to a $\geq$.

Function symbol f **must be bigger than symbol** g Let f and g be function symbols of u and v respectively.

$$gtSym = P_f \geq P_g$$

u **is bigger than every argument of** v Let $argsv$ contain all the arguments of v.

$$gtArgs = \bigwedge_{arg \in argsv} X_{u \geq arg}$$

Combining them all in a single constraint.

$$X_{u>v} \implies orExprs \vee (gtSym \wedge gtArgs)$$

4.2 Results

(a) Find a symbol ordering such that the following inequalities all simultaneously hold, where x, y, z, u, v are all variables, and all other symbols are functions.

Precedence: $f = h = c = a > d = b > g$ Provide the symbol ordering, and give the (human-readable) proof for the third inequality.

```
(RULE) 15. h(g(x,g(u,z)),c(x,y,x,z)) > a(d(x,z),u) by COPY because h >= a and 16, 18
       16. h(g(x,g(u,z)),c(x,y,x,z)) > d(x,z) by COPY because h >= d and 19, 17
       17. h(g(x,g(u,z)),c(x,y,x,z)) > z by SUB because 34.
       18. h(g(x,g(u,z)),c(x,y,x,z)) > u by SUB because 35.
       19. h(g(x,g(u,z)),c(x,y,x,z)) > x by SUB because 36.
       20. d(a(x,y),g(u,v)) > y by SUB because 62.
       21. d(a(x,y),g(u,v)) > u by SUB because 58.
       22. d(a(x,y),g(u,v)) > v by SUB because 60.
       23. d(a(x,y),g(u,v)) > x by SUB because 57.
       34. g(x,g(u,z)) > z by SUB because 59.
       35. g(x,g(u,z)) > u by SUB because 63.
       36. g(x,g(u,z)) > x by SUB because x >= x.
       57. a(x,y) > x by SUB because x >= x.
       58. g(u,v) > u by SUB because u >= u.
       59. g(u,z) > z by SUB because z >= z.
       60. g(u,v) > v by SUB because v >= v.
       62. a(x,y) > y by SUB because y >= y.
       63. g(u,z) > u by SUB because u >= u.
```

(b) Can the inequalities be oriented with any other symbol ordering?

Yes, precedence: $d = f = c > g = b > h = a$

(c) Write a program that, given a term rewriting system, prints YES along with the precedence if it can be ordered using MPO, and that prints NO if it cannot be ordered using MPO.

Check.