Universiteit Leiden

# Futoshiki solving and puzzle design

Niels van Beijnen (s3778053)

Supervisors:
Jonathan K. Vis & Mark J. H. van den Bergh

Bachelor Thesis – Informatica

Leiden Institute of Advanced Computer Science (LIACS)
liacs.leidenuniv.nl

August 23, 2026

**Abstract**

A Futoshiki puzzle is a Latin square completion puzzle with inequality signs between two adjacent cells. To generate well-designed puzzles, we need an efficient solver, an appropriate generation algorithm, and a way to determine the difficulty of a puzzle. We consider and compare three different solving methods: a heuristic backtracking approach, a Boolean satisfiability formulation, and an Integer Linear Programming formulation. A variant of the heuristic backtracking method is the fastest option across all conducted experiments. Next, we discuss a bottom-up and a top-down method for generating puzzles. We recommend the top-down approach as it is more time-efficient and allows us to more easily control the number of hints. Finally, we describe the candidate list strategy and examine how it can be used to assign a difficulty rating to a puzzle. Our rating system produces results similar to those of a prominent source.

# Contents

# 1 Introduction

A Latin square of size $N$ is a square of $N \times N$ numbers, in which each row and column contains the numbers 1 to $N$ a single time in any order. A Latin square completion puzzle is a square of numbers in which some cells are empty. Numbers must be placed in these cells so that a Latin square is constructed. These puzzles usually feature an additional constraint. The most popular example of a Latin square completion puzzle is Sudoku [1], where a Latin square of size 9 should be constructed. The Sudoku square is further divided into squares of size 3, which should also uniquely feature the numbers 1 to 9 in any order as an additional constraint. A lesser known puzzle of this type is the Futoshiki puzzle, created by Tamaki Seto in 2001 [2]. In Japanese, Futoshiki means 'Inequality': a Futoshiki puzzle has a number of inequality ($>$) signs between two cells, meaning that it is specified which one of these cells should have a greater value. Because of the unique additional constraint of Sudoku, its size is restricted to square numbers. Standard Futoshiki puzzles range from 4×4 to 9×9 puzzles. But, unlike for Sudoku, any positive integer $N$ can be used to construct an $N \times N$ Futoshiki puzzle. An example of a 4×4 Futoshiki puzzle is found in Figure 1. Each row and column must contain the numbers 1 to 4. Using the inequality sign in the top left of the puzzle, we know that we must assign the number 4 to the top left cell, as this is the only possible number that is greater than 3. After placing the 4, we see that the third cell in the top row should be assigned the number 2 as this is now the only missing number in the top row.

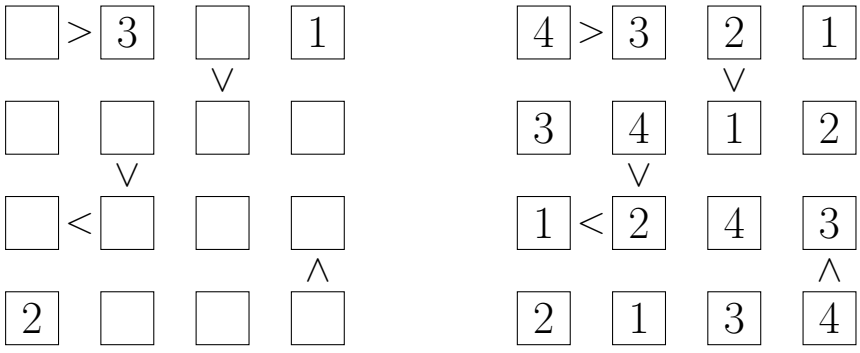


Figure 1: A 4×4 Futoshiki puzzle with 3 given numbers and 5 inequality signs, along with its solution.

A solution to an $N \times N$ Futoshiki puzzle could be any valid Latin square of size $N$. For $N = 5$, there are 161 280 different Latin squares. For $N = 8$, this number is greater than 100 quintillion [3]. Many of these Latin squares are equivalent under symmetry operations. If squares related by such symmetries are identified, the number of different Latin squares is considerably smaller. However, we argue that these symmetrically equivalent squares should be regarded as distinct solutions to a puzzle.

A Futoshiki puzzle has a number of hints to guide solvers in the right direction and to ensure that the puzzle has a unique solution. In Futoshiki, there are two types of hints that can be used to solve a puzzle. There are number hints, where the number of a cell is given, and there are inequality hints, which are placed between two cells to signify that the number in one cell should be greater than the number in the cell across from the inequality sign. As there are $N \cdot N$ cells, there can be any number from 0 to $N^2$ number hints. An inequality can be placed between any two horizontally or vertically adjacent cells, which means that there can be any number from 0 to $2 \cdot N^2 - 2 \cdot N$ inequality hints.

In this research, we are concerned with the design of Futoshiki puzzles, and how to generate well-designed puzzles. An important part of this process consists of being able to algorithmically solve puzzles efficiently.

### 1.1 Related work

Several methods to efficiently solve Futoshiki puzzles have been studied. These include ant colony optimization [4], list coloring [5], and a genetic algorithm [6]. Concerning Futoshiki puzzle design, the number of inequality signs that a puzzle should have was studied in [7]. Besides this, there is little research on Futoshiki puzzle generation and design. To learn about these topics, we can look toward more popular puzzles. For Sudoku, past research includes puzzle generation [8] and estimating the difficulty of puzzles [9].

### 1.2 Problem statement

We aim to discover how to generate well-designed Futoshiki puzzles. A well-designed puzzle is a puzzle that is interesting for a human puzzler to solve. Our criteria for a well-designed Futoshiki puzzle is as follows: firstly, the puzzle must be solvable and have a unique solution. Secondly, the puzzle must be of appropriate difficulty. For an experienced human solver, we want to generate a puzzle that is relatively challenging to solve, as an easy puzzle would be uninteresting to this person. However, an inexperienced puzzler may become frustrated trying to solve a challenging puzzle. Therefore, we must be able to determine the difficulty of a puzzle and generate puzzles of varying difficulty levels. Lastly, for a Latin square completion puzzle, solving the puzzle should involve using its unique additional constraint, rather than only filling in a Latin square. For a Futoshiki puzzle, this means that the inequality constraints should frequently be used to logically determine the number of the next cell, as discussed in [7]. To generate puzzles that meet these criteria, we must answer the following three questions:

1. How can we solve Futoshiki puzzles of various sizes and hint configurations efficiently?
2. How can we generate a large variety of different Futoshiki puzzles?
3. How can we determine the difficulty of a Futoshiki puzzle?

In Section 2, we outline various algorithmic methods for solving Futoshiki puzzles. Section 3 covers puzzle generation approaches. In Section 4, we conduct experiments for the discussed solving and generation methods. We cover difficulty of human puzzle solving in Section 5, and we close by considering an improvement to one of our solving methods.

## 2 Solving Futoshiki puzzles

In order to generate Futoshiki puzzles and judge their difficulty, we need methods to solve a Futoshiki puzzle. For example, we need a solver to ensure that a puzzle is uniquely solvable, and generating a puzzle from scratch potentially involves many calls to this solver. Furthermore, we do not limit ourselves to the standard Futoshiki puzzle dimensions in this research; we consider a

generalized version where the dimension is a parameter. Therefore, we need a solver that efficiently finds a solution, and one that works well not only for small puzzle dimensions, but also for larger ones. In this section, we outline various solving methods.

## 2.1 Complexity of Futoshiki solving

Before diving into specific solving methods, we first consider the complexity of the Futoshiki solving problem. We claim that, given a partially filled-in Futoshiki puzzle, finding an assignment of numbers to all cells such that the Futoshiki puzzle restrictions are met is an NP-complete problem. To prove this claim, we have to show that the problem is a member of the NP complexity class, as well as showing that the problem is NP-hard.

A problem is an element of the NP complexity class when it can be solved by a nondeterministic Turing machine in polynomial time. In other words, problems are in NP when a solution can be verified in polynomial time. In the case of a Futoshiki puzzle, this means that we must be able to determine whether a complete assignment of numbers to cells is a legitimate solution or whether the given assignment is invalid. We assume that the solution consists of a puzzle square where each cell is assigned a number between 1 and $N$ inclusive. We first consider the Latin square restriction: all numbers assigned to the same row or column must be unique. For one row, we can compare each number in the row with all other numbers in the row to determine that they are different. The same applies to the columns. This requires $\Theta(N^2)$ operations for all $N$ rows and $N$ columns, and can therefore be completed in $\Theta(N^3)$ time. Next, we verify the inequality restrictions. A Futoshiki puzzle can feature a maximum of $2 \cdot N^2 - 2 \cdot N$ inequality signs. Each inequality sign can be verified by comparing the numbers in the two cells adjacent to the inequality sign to see whether the inequality holds. Considering that there can be anywhere from 0 to $2 \cdot N^2 - 2 \cdot N$ inequality signs in a puzzle, this restriction is verified in $\mathcal{O}(N^2)$ time. All in all, verifying a solution to a Futoshiki puzzle as described requires $\Theta(N^3)$ time, proving that Futoshiki solving is a member of the NP class.

A problem is NP-hard if all problems in NP can be reduced to it in polynomial time. To prove that the Futoshiki solving problem is NP-hard, we show a reduction from Partial Latin Square Completion (PLSC). A partial Latin square is a Latin square with some number of empty cells. The PLSC problem asks whether the partial Latin square can become a Latin square by assigning numbers to the empty cells. PLSC has been shown to be NP-complete [10], and thus NP-hard, so producing a polynomial reduction from PLSC to Futoshiki ($PLSC \leq_p Futoshiki$) shows that Futoshiki is NP-hard. For this, we must be able to convert every possible instance of PLSC into an instance of Futoshiki in polynomial time, where the Futoshiki instance is solvable if and only if the PLSC instance is solvable. Notice, however, that a Futoshiki puzzle with zero inequality signs is exactly a partial Latin square. Because of this, our reduction becomes simple: given a partial Latin square, construct a Futoshiki instance by copying the square and introducing no inequality constraints. A completion of the partial Latin square exists if and only if the constructed Futoshiki instance has a valid solution, since the only remaining constraints are the uniqueness constraints of the rows and columns. The construction is clearly polynomial, as all that is required is to copy the $N^2$ cells of the square. This proves that $PLSC \leq_p Futoshiki$, and thus that Futoshiki is NP-hard.

Since Futoshiki is in NP and is NP-hard, we conclude that Futoshiki solving is an NP-complete problem. Having established the complexity class, we now consider various algorithms for solving Futoshiki puzzles.

## 2.2 Backtracking

The most straightforward way to algorithmically solve constraint satisfaction problems such as Futoshiki puzzles is by using a backtracking algorithm [11]. A backtracking algorithm for the Futoshiki puzzle works as follows: starting at the top left cell, we work our way to the right and down. When we encounter an empty cell, we try to place each number from 1 to $N$ in the cell in this order. After placing a number, we check if our puzzle is still valid with regard to the restrictions of the Futoshiki puzzle rules. If the puzzle is valid after adding the number to this cell, we move to the next empty cell and place a number there. If the puzzle has become invalid, we try the next number for the current cell. If we have tried each number for the current cell and all of them lead to an invalid puzzle state, we go back to the previous cell to try the next number there. Once we place a number in the last empty cell and the puzzle is still valid, we know that we have found a solution. At that point, we can choose to stop after having found a single solution, or we can choose to continue in the same manner to eventually find the total number of different possible solutions. If we have tried all possible valid combinations of numbers for all cells and still have not found a solution, we conclude that the puzzle is in an invalid state and is unsolvable.

**Algorithm 1** Recursive backtracking solver to find all solutions

**Pre:** Start at top-left cell of a puzzle with at least one empty cell. A global variable *solutions* is initialized with value 0.

**Post:** Variable *solutions* holds the total amount of different solutions to the puzzle.

```
if all cells filled then
    solutions += 1
    return
end if

if current cell is not empty then
    move to next cell
end if

for num from 1 to N do
    current cell = num
    if puzzle still valid then
        move to next cell
    end if
end for

current cell = empty
```

Backtracking is commonly implemented using recursion. A simplified version of this method is shown in Algorithm 1. The algorithm can be modified to stop searching after finding a single solution. A significant part of the algorithm lies in verifying that the puzzle is still valid after placing a number in a cell. Until now, we have skipped over this part: how do we actually check if the puzzle is still valid? To do this, we need to analyze the puzzle to verify that no rules of a

Futoshiki puzzle have been violated: there are no duplicate numbers in rows and columns, and neighboring numbers are in accordance with the specified inequality signs.

In Section 2.1 we discuss a naive method to verify the validity of a puzzle in $\Theta(N^3)$ time. This approach considers the entire puzzle, however, when running the backtracking algorithm, only a single number is placed in a single cell between consecutive validity checks. Using this observation, we notice that checking the entire puzzle for each validity check is redundant; it suffices to solely check the corresponding row and column of the cell that was just filled in, as well as neighboring inequality constraints. Checking the entire puzzle requires us to look at all $N^2$ cells, and $2 \cdot N^2 - 2 \cdot N$ inequality signs in the worst case. Instead, for each validity check we only require to look at the $N - 1$ cells in the row of the cell that we just placed a number, another $N - 1$ cells for the column of the cell, and a maximum of 4 neighboring cells for the inequality signs. To check for duplicates in a row or column, we could use a standard data structure like an unordered set. However, this data structure is commonly implemented using hashing, and considering that the numbers always lie in the range $[1, N]$, an even better option is to use a Boolean array of size $N$ where each bit with index $k$ represents whether we previously encountered the number $k$. This removes the often large constant factor that standard library implementations of hashing typically require, and replaces it with instant array indexing. In total, we are able to limit the validity check to an order of $\Theta(N)$ operations, instead of the $\Theta(N^2)$ operations required to check the entire puzzle. Furthermore, it is not necessary to recompute the Boolean array for encountered numbers for rows and columns each time. Instead, we can store an array for each row and column and adjust these values when adding, changing, or removing a number from a cell. With this, the time complexity of the validity check is further reduced to $\Theta(1)$. We do, on the other hand, need extra memory space for this optimization, as each row and column stores $N$ bits, requiring $\Theta(N^2)$ memory space. However, this memory requirement is smaller than the memory required to store the puzzle itself, as $N^2$ integers need to be stored, with $\log_2(N)$ bits needed to represent the integers for each cell. This means that memory requirement is still limited by the larger $\Theta(N^2 \cdot \log_2(N))$ factor needed to represent the puzzle.

Running this recursive function on a modern computer for large puzzle instances reveals a problem. Considering the exponential growth in the number of possibilities for larger puzzles, the recursion depth quickly becomes too much. To circumvent this issue and provide the ability to solve all reasonably sized puzzles, we rewrite the function to make use of a stack to keep track of the solving state. This, however, introduces a new problem. After trying all possibilities for some cell and returning to the previous cell, we need to know whether the number that this previous cell holds was given, or whether it was initially empty and we filled it in during the solving process. We need to distinguish between these cases because we do not want to change given numbers as doing so leads to finding invalid solutions. On the other hand, if we filled the number in ourselves, we want to be able to change the number in the cell to keep trying new possibilities. In the recursive function, this distinction is made implicitly by returning to the function call of some state. For the iterative function, distinguishing these cases can be done by storing a bit for each cell: the bit is set when the number in the cell is given, and not set if the cell was initially empty. A simplified version of the iterative algorithm is outlined in Algorithm 2. The algorithm becomes significantly more complicated compared to the recursive version, but using a stack allows solving puzzles of much larger sizes.

All in all, the backtracking method is quite straightforward. We try to fill the empty cells with all possible valid combinations of numbers in some arbitrary order until the puzzle becomes invalid

**Algorithm 2** Iterative backtracking solver to find all solutions

**Pre:** The puzzle has at least one empty cell. A global variable *solutions* is initialized with value 0.
**Post:** Variable *solutions* holds the total amount of different solutions to the puzzle.

```
stack = {top left cell}
while stack not empty do
    current cell = stack.pop
    if current cell was given then
        push next cell onto stack
    else if current cell holds value N then          ▷ Tried all possibilities for current cell
        current cell = empty
    else
        if current cell is not empty then                    ▷ Try next untried number
            start num = value of current cell + 1
        else
            start num = 1
        end if
        for num from start num to N do
            current cell = num
            if puzzle valid then                     ▷ Found number to try: stop for-loop
                break
            end if
        end for
        if puzzle valid then
            if all cells filled then
                solutions += 1
            else
                push current cell onto stack     ▷ Save current state to try other possibilities later
                push next cell onto stack
            end if
        else
            current cell = empty
        end if
    end if
end while
```

or a solution is found. The downside of this naive solving strategy is that it requires considerable computation time to solve most puzzle instances, and struggles especially on larger puzzles. To efficiently solve and generate Futoshiki puzzles, we need a better option.

## 2.3 Heuristics

Considering the role of a solver in generating a puzzle, we mainly want the solver for two purposes. Firstly, while generating a puzzle, we need to make sure it is actually solvable. For this, we need to find at least one solution, or conclude that no solution is possible. Secondly, we want to ensure that a puzzle is uniquely solvable. A puzzle should have precisely one solution, and not more. Notice that for both of these tasks, we are not interested in the precise number of solutions to a puzzle. Instead, it suffices to know whether the puzzle has zero, one, or more than one solution. For our purposes, we never have to continue searching after finding two solutions. This means that in most cases, rather than an algorithm that finds all solutions quickly, it is beneficial for us to employ an algorithm that finds *one* solution quickly.

The naive backtracking algorithm arbitrarily chooses the topmost leftmost unfilled cell to place a number in. We want to find out whether the order of filling empty cells has an effect on the performance of the backtracking algorithm, as changing the order may allow us to reach a solution earlier. We are specifically interested in the number of possibilities we need to try before finding a solution: we want to make decisions in such a way that we encounter a solution early on, avoiding trying a huge number of possibilities until one eventually fits.

We could try to change the order to some other predetermined order of the cells, but any other arbitrary order is only be beneficial for some specific puzzle configurations, and leads to an increase in execution time for other configurations. We want to utilize a more thought-out approach which improves performance for general puzzle configurations. To do this, we need to make decisions based on the specific puzzle that is being solved. In other words, while running the algorithm, we need to dynamically choose which cell to select next based on the current state of the puzzle. This heuristic approach is used in various problems, for example in [12]. Heuristics for puzzles similar to Futoshiki such as Sudoku have also been studied [13]. In case of Futoshiki, how do we decide which cell to visit next?

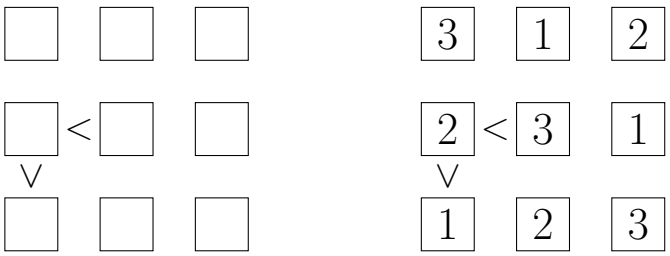


Figure 2: A small Futoshiki puzzle with its unique solution shown on the right.

Consider the small puzzle in Figure 2. Despite having only two inequalities and no given numbers, the puzzle has a unique solution. We see that the top left cell must be a 3. However, our backtracking solver starts at the top left cell by placing the numbers from 1 to 3 in that order. It starts by assigning a 1 to the top left cell, and then keep assigning numbers to other cells until it is faced with a contradiction, in which case it backtracks. After unsuccessfully trying all combinations where the

top left cell contains a 1, it places a 2 there and repeats this process. After more backtracking, it finally places a 3 in the top left cell.

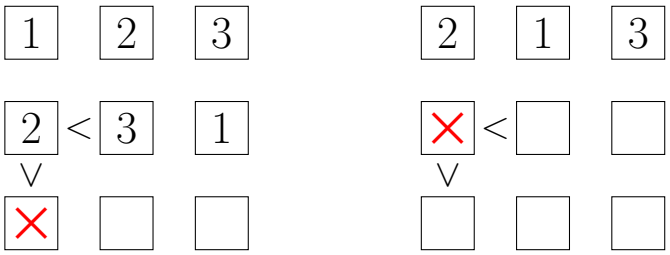


Figure 3: States in which the standard backtracking method encounters a contradiction and must backtrack.

A part of the standard backtracking process is shown in Figure 3. We see that many numbers are placed before encountering a contradiction and many possibilities must be tried before finding the solution. If, instead of arbitrarily starting from the top left cell, we start from the cell that we have most information about, then we would look at the leftmost cell in the second row, as it is involved in two inequality constraints. We see that it should be smaller than the cell to the right, meaning it cannot be a 3, and larger than the cell to the bottom, meaning it cannot be a 1. The cell must be assigned a 2. Knowing that the cell to the bottom is smaller, we assign it a 1. Now, the only missing number in its column is 3, so we know that the top left cell must be a 3. Visiting the cells in this order allows us to avoid any backtracking before placing the correct number in the top left cell. This process is shown in Figure 4.

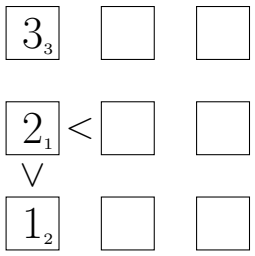


Figure 4: The solving progress after three steps when choosing the cell with the most information. The subscript of a number shows in which order the cells were visited.

Continuing to visit the empty cell with the most information allows us to solve the entire puzzle without any backtracking. Because of this, we study the effect of the following heuristic: when choosing some cell to place a number in, choose the cell that has the least amount of possibilities left, arbitrarily breaking ties. Considering that solving most puzzles requires making an exponential number of decisions, we choose to keep our heuristic simple: a number is considered a possibility for a cell if it is not ruled out directly by the rules of Futoshiki, meaning that the number does not already appear in the row or column of the cell and does not violate an inequality constraint. Notice that this addition requires only a subtle change to the backtracking algorithm: for an iterative version, the pseudocode from Algorithm 2 stays mostly the same. The only change needed is to push onto the stack the next cell that is chosen based on the heuristic, rather than the next cell from top left to bottom right, as well as initializing the stack with the appropriate cell.

This heuristic approach does come with an extra cost: now, each time a choice has to be made for which cell to fill in next, the possibilities for each cell have to be known. This can be calculated directly for each cell each time a decision has to be made. Doing this naively, we visit all $N^2$ cells and for each cell we check whether each of the $N$ numbers is possible. To check this, we would look at each cell in the row and column of the cell, as well as checking adjacent cells in case of inequality signs, visiting $2 \cdot N - 2$ cells. This amounts to $N^2 \cdot N \cdot (2 \cdot N - 2)$ operations, resulting in a time complexity of $\Theta(N^4)$. Using Boolean arrays to store the available numbers per row and column reduces the complexity of the heuristic to $\Theta(N^3)$, as it removes the need to visit the $2N - 2$ cells in the same row and column for each number, only leaving a maximum of 4 cells to visit for the inequality constraints.

We can further reduce the complexity by explicitly storing the possible values for each cell, in which case each cell has a bit-mask for the possible numbers, as well as an integer denoting the total amount of possibilities, or set bits in the bit-mask. With this, choosing a cell requires $\Theta(N^2)$ operations as each cell is visited once to find the minimum value. This does require $N$ bits in memory for all cells, which means memory requirement increases to $\Theta(N^3)$. On top of that, updating the possibilities requires significant additional logic during the puzzle solving process. Each time the number in a cell is added, changed, or removed, all values for the cells in its row and column have to be updated, making the algorithm significantly more complicated from an implementation standpoint. With this optimization, we choose some cell with our heuristic in $\Theta(N^2)$ time, and then we place $N$ different numbers in this cell in the worst case. Notice that each time we try a different number, the values of the cells in the corresponding row and column are updated, amounting to $\Theta(N^2)$ operations. This shows that even with the added need to update values many times, the efficiency of the heuristic function still benefits from this change, especially considering the usually vast number of times it will be used to find a solution. An experiment to compare the standard backtracking approach to the heuristic approach is conducted in Section 4.1.

## 2.4 Boolean satisfiability

The Boolean satisfiability problem, abbreviated as SAT, is a well-known NP-complete problem. SAT was the first problem proven to be NP-complete [14] and remains a recurring element of many complexity proofs. Because it is such a popular problem, many have attempted to create efficient solvers [15], and there are annual SAT competitions where experts gather to attempt to optimize SAT solving efficiency [16]. As a result, SAT solving methods have become increasingly complicated and efficient, which is why problems in NP can often be solved efficiently by translating them into an instance of SAT and using a SAT solver to find an answer.

The SAT problem asks whether there exists an interpretation of Boolean variables that satisfies a given Boolean expression. If no satisfying interpretation is possible, the formula is said to be unsatisfiable. SAT instances are usually represented using Conjunctive Normal Form (CNF), both for convenience and because many techniques used in SAT solvers benefit from this format in terms of implementation efficiency. Therefore, to use these solvers to solve a Futoshiki puzzle instance, we need to represent a Futoshiki instance as a Boolean expression in CNF. The puzzle must be represented as an expression using Boolean variables, and this expression must be satisfiable if and only if a solution exists for the puzzle.

We first need a way to represent the numbers in the cells of the puzzle using Boolean variables. To allow each cell to have a value from 1 to $N$, we introduce $N$ variables per cell. Variable $x_{i,j}^k$ represents

the number $k$ in row $i$ and column $j$. We say that the number $k$ should be placed in cell $(i, j)$ if and only if variable $x^k_{i,j}$ is `True`. With these variables, we need to represent the rules of Futoshiki in a Boolean expression: every cell must be assigned a number, rows and columns must contain unique numbers, and inequality constraints must be satisfied.

We first enforce that a cell must not contain multiple numbers. For all variables that represent a number in the same cell, only one of these variables must be `True` for each cell. Consider that we have variables $x^1_{1,1}$ and $x^2_{1,1}$. We must prevent both of these variables from being `True` at the same time, as this would mean that cell $(1, 1)$ contains both a 1 and a 2. To do this, we add the clause $\neg x^1_{1,1} \vee \neg x^2_{1,1}$. To satisfy this clause, either $x^1_{1,1}$ must be `False` or $x^2_{1,1}$ must be `False`; they cannot both be `True` at the same time. To ensure that a cell cannot contain multiple numbers, we add these clauses for every pair of two valid numbers for every cell:

$$\bigwedge_{i=1}^{N} \bigwedge_{j=1}^{N} \bigwedge_{k=1}^{N-1} \bigwedge_{\ell=k+1}^{N} \left(\neg x^k_{i,j} \vee \neg x^\ell_{i,j}\right).$$

This creates $N^2 \cdot N \cdot \frac{N-1}{2} = N^3 \cdot \frac{N-1}{2}$ clauses with two literals each. However, it remains possible that all variables for a cell are set to `False`, meaning that this cell would remain empty. Before fixing this, we consider the Latin square constraint: every number should appear in every row and every column. For our Boolean expression, this means that for all rows $i$ and all numbers $k$, $x^k_{i,j}$ must be `True` for at least one column $j$:

$$\bigwedge_{i=1}^{N} \bigwedge_{k=1}^{N} \bigvee_{j=1}^{N} x^k_{i,j}.$$

These $N^2$ clauses of $N$ literals guarantee that all rows must contain every number. The same must be done for the columns:

$$\bigwedge_{j=1}^{N} \bigwedge_{k=1}^{N} \bigvee_{i=1}^{N} x^k_{i,j}.$$

As with the rows, this constraint adds $N^2$ clauses of $N$ literals to our Boolean formula. Considering that that there are $N$ possible numbers, these clauses ensure that every row and column has at least $N$ variables set to `True`: at least one for each number. From the first set of clauses, we force each cell to have no more than one variable set to `True`. Since a row or column contains $N$ cells, we know that no more than $N$ variables in a row or column are set to `True`. Combining this with the clauses we add for the Latin square constraint, which force that *at least* $N$ variables are `True` in a row or column, we know that a row or column must have precisely $N$ `True` variables. By adding these rules, each cell is forced to have exactly one number, thus there is no need to add other clauses to explicitly prevent a cell from having zero `True` variables.

Now, the Latin square constraints are satisfied. All that remains is to take care of the inequality constraints. Say that cell $(1, 1)$ is greater than cell $(1, 2)$. Now say that cell $(1, 1)$ contains the number 4. Then we know that all variables $x^k_{1,2}$ with $k \geq 4$ should be `False`: cell $(1, 2)$ cannot contain a number greater than 4. For all inequality signs where cell $(i, j)$ should have a number greater than the number in cell $(p, q)$, we add the following clauses:

$$\bigwedge_{k=1}^{N-1} \bigwedge_{\ell=k+1}^{N} (\neg x^k_{i,j} \vee \neg x^\ell_{p,q}).$$

This further adds $N \cdot \frac{N-1}{2}$ two-literal clauses to our expression for each inequality sign. Our expression now produces a solution that conforms to the rules of Futoshiki puzzles. To solve partially filled-in puzzles, the solution must be consistent with the given numbers. For this, we can simply state that if cell $(i, j)$ already contains the number $k$, variable $x_{i,j}^k$ must be `True`, adding a one-literal clause for each number hint.

The formula is now complete: the produced Boolean expression is satisfiable if and only if the puzzle has a solution. In total, we use $N^3$ variables and create $\Theta(N^4)$ clauses and $\Theta(N^4)$ literals for this translation. Now, we pass this expression to a SAT solver. A solver returns *unsatisfiable* if there is no solution to our puzzle. Else, it provides a list of values `True` or `False` for each variable in a satisfying assignment. We use this list to complete the puzzle, placing the number $k$ in cell $(i, j)$ if and only if the solver tells us that variable $x_{i,j}^k$ is `True`. If there are multiple solutions to a puzzle, then the solver provides any satisfying assignment. In this case, different solvers can provide different assignments, and for most solvers, it is not specified which solution it finds. However, to generate a puzzle, we need to know whether it is uniquely solvable, i.e. whether there are zero, one, or more than one solution. How can we enumerate puzzle solutions with Boolean expressions if a solver just provides some non-specified solution? Upon finding a solution, we can add what is called a blocking clause. A blocking clause is a clause that blocks finding a specific solution. After receiving a solution from our solver, we can add a blocking clause for this specific solution to our expression and ask the solver to solve this new expression. This forces it to find a new solution, or claim *unsatisfiable* if no other solutions are available. Given some assignment of an expression, the blocking clause is the disjunction of the negation of all of these assignments. Suppose that we have some solution to a Boolean expression consisting of three variables $x$, $y$ and $z$, with our satisfying assignment $x \wedge \neg y \wedge z$. Then, our blocking clause becomes $\neg x \vee y \vee \neg z$. Now, a new assignment must differ in at least one place from this assignment. In the Futoshiki case, we add a single clause of $N^3$ literals for each solution we find. To find all solutions to a puzzle, we repeatedly solve and add a blocking clause to our CNF expression until the expression has become unsatisfiable, at which point we know we have found all of the possible solutions. For near-empty puzzles with millions of possible solutions, SAT solving to find all solutions is not viable as adding $N^3$ literals for each solution quickly becomes infeasible. For puzzle generation however, where it is never necessary to find more than two solutions, solving through SAT may be very effective.

## 2.5 Integer Linear Programming

In the SAT problem, variables in a Boolean expression can only take values `True` or `False`. Because of this, there is no direct way to represent numbers, which is why every possible number for every cell must be represented by a specific variable, requiring $N^3$ variables in total. Integer Linear Programming (ILP) is a mathematical optimization or feasibility technique used to find a solution to a problem while satisfying a set of linear constraints [17]. The effectiveness of ILP for Futoshiki solving has been studied in [18], however, enumeration is not discussed and the experiments are limited. In ILP, variables are restricted to be integers, meaning that a solution to our Futoshiki puzzle can be represented by using only $N^2$ variables. Like SAT, ILP is proven to be NP-complete [19], and numerous efficient solvers have been created. To study the effectiveness of ILP for solving Futoshiki puzzles, we translate a Futoshiki instance into an ILP instance and request a modern ILP solver to find a solution.

Linear constraints are of the form $A \leq B$, where $A$ and $B$ are linear equations consisting of

variables and constants. Given some variables and constraints, an ILP solver assigns values to the variables such that all the constraints are met, or returns *infeasible* if no solution exists. We have to formulate a Futoshiki puzzle as a set of linear constraints such that the set of constraints can be satisfied if and only if there exists a valid solution to the Futoshiki puzzle. This means that that we have to find a way to model the Latin square constraints and inequality constraints as a set of linear constraints.

We declare $N^2$ integer variables: one variable for each cell of the puzzle, denoted as $x_{i,j}$ for the cell in row $i$ and column $j$. Each variable must have a value between 1 and $N$ inclusive, so we add two constraints for each variable: $1 \leq x_{i,j}$, and $x_{i,j} \leq N$. This requires a total of $2 \cdot N^2$ constraints. Each given number is also encoded by using two linear constraints. If cell $(i, j)$ contains the number $k$, we introduce constraints $x_{i,j} \leq k$ and $k \leq x_{i,j}$, thus forcing $x_{i,j} = k$.

Concerning Futoshiki rules, we start by considering the inequality constraints. In ILP, this is trivial: for all inequality signs where cell $(i, j) <$ cell $(p, q)$, we add a single linear constraint of the form $x_{i,j} \leq x_{p,q} - 1$. In the worst case, this adds $2 \cdot N^2 - 2 \cdot N$ linear constraints, as this is the maximum number of inequality signs a puzzle can have.

The Latin square constraint is significantly more complicated in ILP. To force all numbers in a row or column to be unique, we need a way to force two variables to be different. While forcing two variables to be equal, as we do for given numbers, is simple, forcing two variables not to be equal is not as straightforward. For variables $A$ and $B$ to be different, either $A > B$ or $A < B$. However, such a disjunction cannot be directly represented with linear constraints. To encode this, we must introduce a new variable $y$. This variable serves as a Boolean variable indicating whether $A > B$ or $A < B$. We implicitly create this Boolean variable by introducing two constraints to limit its value: $0 \leq y$ and $y \leq 1$. Now we add two constraints to force that $A \neq B$. The first is $A - B - M \cdot y \leq -1$, where $M$ is a large constant value. Notice that if $A < B$, then the constraint is satisfied for both $y = 0$ and $y = 1$. However, if $A \geq B$, then $y$ must take value 1 to satisfy the constraint. The second constraint that we add is $B - A + M \cdot y \leq M - 1$. If $A > B$, this constraint is always satisfied with a non-negative $M$, regardless of the value of $y$. However, if $A \leq B$, then $y$ must take the value 0 to satisfy the constraint. Combining the two constraints, we see that in the case where $A = B$, $y$ must take the value 1 to satisfy the first constraint, and 0 to satisfy the second constraint. Both cannot be possible at the same time, so an assignment where $A = B$ is always infeasible. A case distinction is shown in Table 1, outlining that this formulation works as intended.

Table 1: Verification of the constraints forcing $A \neq B$.

| **Case** | $\mathbf{A < B}$ | $\mathbf{A > B}$ | $\mathbf{A = B}$ |
|---|---|---|---|
| $A - B - M \cdot y \leq -1$ | $y = 0 \vee y = 1$ | $y = 1$ | $y = 1$ |
| $B - A + M \cdot y \leq M - 1$ | $y = 0$ | $y = 0 \vee y = 1$ | $y = 0$ |
| Value of $y$ | $y = 0$ | $y = 1$ | $y = 0 \wedge y = 1$: **Infeasible** |

We need to set a correct value for $M$. The first constraint, $A - B - M \cdot y \leq -1$, should always hold if $y = 1$. Rewriting the constraint with $y = 1$ gives $A - B - M \leq -1$. Rearranging yields $A - B + 1 \leq M$, which is equivalent to $A - B < M$. We want this to be true in all cases, so $M$ should specifically be greater than the maximum value of $A - B$. In Futoshiki, the greatest number that a cell can hold is $N$, and the lowest value is 1. This means that $N - 1 < M$; $M$ should

be at least $N$. For the second constraint, $B - A + M \cdot y \leq M - 1$ should always hold if $y = 0$. Rewriting gives $B - A \leq M - 1$, or $B - A < M$. Similar to the first constraint, $M$ should be greater than the maximum possible value of $B - A$, meaning that for Futoshiki, $M \geq N$ also holds for this constraint. In general, $M$ should be greater than the maximum possible difference between $A$ and $B$; the maximum possible value of $|A - B|$.

Considering that all cells in the same row or column must be different to satisfy the Latin square constraint, we must force inequality for all variable pairs $x_{a,j}$ and $x_{b,j}$ (cells in the same row) and all pairs $x_{i,a}$ and $x_{i,b}$ (cells in the same column) where $a \neq b$ for all $1 \leq i, j \leq N$. With this technique, we have to add a variable and the corresponding four linear constraints (two to declare a Boolean variable and two to force inequality) for every pair of cells in a row or column. For one row, there are $N \cdot \frac{N-1}{2}$ unordered pairs of cells, thus adding $N \cdot \frac{N-1}{2}$ variables and $2 \cdot N \cdot (N-1)$ constraints. For $N$ rows and $N$ columns, the total becomes $N^2 \cdot (N-1)$ variables and $4 \cdot N^2 \cdot (N-1)$ constraints. Even though only $N^2$ variables are needed to represent the puzzle compared to the $N^3$ variables needed for SAT, the extra variables required to encode the Latin square constraint bring the total number of variables to $N^2 \cdot (N-1) + N^2 = N^3 - N^2 + N^2 = N^3$, the exact same number of variables required for SAT. To encode the Futoshiki rules, we need $4 \cdot N^2 \cdot (N-1) = 4 \cdot N^3 - 4 \cdot N^2$ linear constraints for the Latin square rule, $2 \cdot N^2$ constraints to limit the value of each variable, one constraint for each inequality hint and two constraints for each number hint, totaling to $4 \cdot N^3 - 2 \cdot N^2 + 2 \cdot |\text{number hints}| + |\text{inequality hints}|$ constraints.

In ILP, an objective function is often used to help find the best solution to the problem. The objective function is a linear function that is minimized or maximized and guides the solver in the right direction. However, for Futoshiki solving and puzzle generation there is no best solution, as we aim to find any feasible solution. Furthermore, no helpful linear function can be constructed. The values of the variables lie in the range $[1, N]$, and the sum of all variables in a row or column, as well as the sum of all numbers in the entire puzzle, is always the same. This means that our only option would be to minimize or maximize some specific set of cells. Imagine that cell $(i, j)$ is currently empty, but should hold number 1 in the unique solution. In this case, minimizing cell $(i, j)$ would allow the solver to find the solution earlier. However, if cell $(i, j)$ should contain the value $N$ instead, then minimizing the value of the cell would lead to slower solving, and instead maximizing the value of the cell would be beneficial. In general, choosing some cells to minimize or maximize is only beneficial in some specific puzzle instances, and is disadvantageous for other instances. Because of this, no objective function is utilized to solve Futoshiki puzzles with ILP.

With this formulation, we are able to convert Futoshiki puzzle instances into ILP instances that have a feasible solution if and only if the Futoshiki puzzle has a valid solution. We are now able to solve Futoshiki puzzles by feeding the produced variables and set of linear constraints to an ILP solver, which assigns a value to each variable such that the set of constraints is satisfied, or answer *infeasible* if this is not possible. To produce a Futoshiki solution, we simply take the assigned value of variable $x_{i,j}$ and place it in cell $(i, j)$ for each cell.

Next, we discuss how to enumerate puzzle solutions using ILP. After finding some solution, we must add linear constraints based on the encountered solution to our ILP model such that this solution is considered invalid during the next solving cycle. Two solutions are different if at least one of the $N^2$ cells holds a different value in both solutions. In practice, a solution to a Futoshiki puzzle must differ in at least four places from another solution to be considered different. Say we have a valid solution to a puzzle where the top-left cell contains the number 1. If we change the number of this cell to the number $k$ with $k \neq 1$, then the top row and the left column do not

contain the number 1 anymore, and both contain the number $k$ twice. If cell $(1, j)$ and cell $(i, 1)$ contain the number $k$, we can change both cells to have the value 1. This causes row $i$ and column $j$ to have two cells with the number 1, and no cells with the number $k$. In the best case, cell $(i, j)$ contains the number 1, and changing this to $k$ creates a valid and different solution. Therefore, we know that two solutions with at least $N^2 - 3$ cells in common are the same solution, as the last three cells are fixed by the Latin square constraints. To find a new solution, it must have at most $N^2 - 4$ cells in common with any previous solution.

We count the number of cells that have the same value by using a trick similar to the constraints introduced to force two cells to be different. Again using values $A$ and $B$ as an example, we declare a Boolean variable $p$ and introduce constraints $A - B - M \cdot p \leq 0$ and $B - A + M \cdot p \leq M - 1$. The only difference from the set of constraints forcing two numbers to be different is that the term $A - B - M \cdot p$ in the first constraint should be $\leq 0$, instead of $\leq -1$. This now means that if $A = B$, variable $p$ can take either value 0 or 1 for the first constraint, while the second constraint still forces $p = 0$. Now we add another Boolean variable $q$, with the same constraints, where $A$ and $B$ are flipped: $B - A - M \cdot q \leq 0$ and $A - B + M \cdot q \leq M - 1$. Now, if $A < B$, $p = 0$ and $q = 1$. If $A > B$, $p = 1$ and $q = 0$, and if $A = B$, then $p = q = 0$, as shown in Table 2. If we add one last Boolean variable $y = 1 - p - q$, then $y = 1$ if and only if $A = B$. To count all cells that are equal to a previous solution, we add these three Boolean variables where $A = x_{i,j}$ and $B = solution_{i,j}$ for all $1 \leq i, j \leq N$. To ensure that a new solution is found, the sum of the $y$ variables for each cell must be less than $N^2 - 3$: at least four cells must be different. The constraint $sum(y) \leq N^2 - 4$ is added.

Table 2: Verification of the constraints forcing $y = 1$ if and only if $A = B$.

| **Case** | $\mathbf{A < B}$ | $\mathbf{A > B}$ | $\mathbf{A = B}$ |
|---|---|---|---|
| $A - B - M \cdot p \leq 0$ | $p = 0 \vee p = 1$ | $p = 1$ | $p = 0 \vee p = 1$ |
| $B - A + M \cdot p \leq M - 1$ | $p = 0$ | $p = 0 \vee p = 1$ | $p = 0$ |
| Value of $p$ | $\mathbf{p = 0}$ | $\mathbf{p = 1}$ | $\mathbf{p = 0}$ |
| $B - A - M \cdot q \leq 0$ | $q = 1$ | $q = 0 \vee q = 1$ | $q = 0 \vee q = 1$ |
| $A - B + M \cdot q \leq M - 1$ | $q = 0 \vee q = 1$ | $q = 0$ | $q = 0$ |
| Value of $q$ | $\mathbf{q = 1}$ | $\mathbf{q = 0}$ | $\mathbf{q = 0}$ |
| $\mathbf{y = 1 - p - q}$ | $\mathbf{y = 0}$ | $\mathbf{y = 0}$ | $\mathbf{y = 1}$ |

To prevent the solver from finding the same solution, we need six constraints for each cell to restrict the three new variables to Boolean values. Furthermore, we need the four constraints for $p$ and $q$ as shown, two more constraints to force $y = 1 - p - q$, adding twelve in total per cell. For each found solution, this adds $3 \cdot N^2$ variables and with one last constraint added to limit the sum of all $y$, $12 \cdot N^2 + 1$ total constraints are added to the model. To find all solutions to a puzzle, we ask the solver to find a solution to our set of linear constraints, add exclusion constraints for the found solution, and repeat this process until the model is infeasible, at which point all solutions have been discovered. Similar to SAT, invalidating past solutions greatly increases both memory and computation requirements for subsequent solving as a potential solution must be compared to all previous solutions, suggesting that it is not the best method to find all solutions to a near-empty puzzle. However, the method may excel at finding one or two solutions, or concluding that there are no solutions, as required by the generation process.

In this section, we have outlined four solving strategies: a standard backtracking method, a backtracking method using a heuristic function, a SAT formulation, and an ILP formulation. For each algorithm, the intuition is shown, the method is explained and optimizations are considered. Examining the performance of each method will show which solving strategies are preferable in which cases. For this purpose, many puzzles of different sizes, different numbers of hints and different initial configurations are needed to provide a conclusive comparison. Therefore, before we are able to compare our solving methods, we must be able to randomly generate a large number of puzzles.

# 3 Generating Futoshiki puzzles

We want to create puzzles with randomly generated configurations that are interesting to solve for a human puzzler. These puzzles should have a unique solution, an appropriate number of hints, and a difficulty rating. Difficulty is discussed in Section 5. This section outlines and compares two approaches for generating a puzzle at random, and how to guarantee that this randomly generated puzzle is uniquely solvable.

Our first approach is a bottom-up generation: starting from an empty puzzle square, inequality hints and number hints are randomly added until it is uniquely solvable. The second approach is top-down: starting from a full Latin square, inequality hints are added and numbers are removed as long as the puzzle remains uniquely solvable.

Before discussing the generation process, let us consider the inequality hints. A Futoshiki puzzle can contain any number of inequality signs in the range $[0, 2 \cdot N^2 - 2 \cdot N]$. The number of inequality signs for a proper Futoshiki puzzle was studied in [7]. The author claims that the number of inequality signs should be an intermediate amount. Considering these signs are what make a Futoshiki puzzle unique, we do not want too few of them. Instead, we want the solver to frequently have the opportunity to use an inequality hint to make progress while solving a puzzle. There also should not be too many inequality signs, as this restricts the puzzle too much and trivializes solving. Based on their findings, we generate puzzles with the number of inequality signs in the range $[N, N^2]$. This allows for considerable variety in the puzzles created, while still adhering to the puzzle design principles mentioned in [7].

## 3.1 Bottom-up approach

Starting from an empty puzzle, we wish to add inequality hints and number hints such that the created puzzle is uniquely solvable. We first consider the number of inequality hints. As established, our aim is to create puzzles with the number of inequality hints in a specific range. Therefore, we add the inequality hints before adding number hints. This allows us to better control the number of inequality hints: because the puzzle is only restricted by the inequalities, we can keep adding them either until the puzzle is uniquely solvable, or until we have added as many as we want. We choose a random number, say $r$, in the range $[N, N^2]$. and add $r$ inequality signs to the puzzle. We randomly choose one of the empty inequality positions, and randomly select either $<$ or $>$ for an inequality between two cells in the same row, or select $\vee$ or $\wedge$ for a vertical inequality. During this process, we have to be careful that an added inequality hint does not cause the puzzle to become unsolvable. A puzzle with only inequality hints can quickly become unsolvable. For example, an

inequality chain could loop, meaning that a number in a cell should be greater than itself. An example of this is shown in Figure 5.

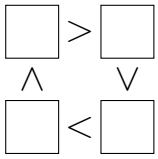

Figure 5: A placement of inequality hints that form a loop, making a puzzle unsolvable.

Because a puzzle could become unsolvable after adding an inequality, we must use one of our solving methods from the previous section to verify that the puzzle still has a valid solution after adding each inequality. If the puzzle has become invalid, we must remove the last inequality that we added. Notice, however, that in this case, we can simply flip the inequality hint at this position to retain a valid puzzle. Imagine that the puzzle is invalid if $A > B$. Considering that inequalities are only placed between numbers in the same row and column, we can also say that $A$ cannot equal $B$. If $A > B$ is invalid and $A = B$ is invalid, then we know that $A < B$. Therefore, upon placing some inequality hint that makes our puzzle unsolvable, we do not have to reselect a position, but we can simply invert the sign without having to verify its validity. After adding the inequality signs, it is possible that the puzzle already has a unique solution. Interestingly, this can occur even after adding only a small number of inequalities. Examples for $N = 4$ and $N = 5$ are shown in Figure 6.

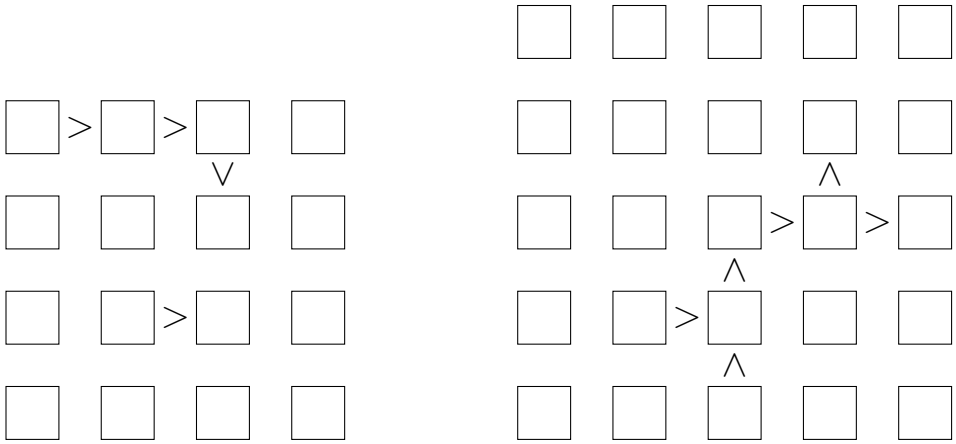

Figure 6: A 4×4 and a 5×5 puzzle that have no number hints and few inequality hints, yet have a unique solution.

If, after adding the desired number of inequalities, the puzzle does not yet have a unique solution, then we must add number hints until it does. To do this, we randomly select a cell and place a random number in the range $[1, N]$ in it. If the puzzle has become unsolvable, we remove the number and try again. If after adding the number the puzzle still has multiple valid solutions, we continue the process until a single solution remains. A problem with this is that in an unfortunate case, almost all cells need to be filled before the puzzle becomes uniquely solvable. An example of a randomly generated puzzle that is not yet uniquely solvable is shown in Figure 7. In the second and fourth rows, the numbers 3 and 4 are missing. Placing these numbers in either order in the second row leads to a valid solution.

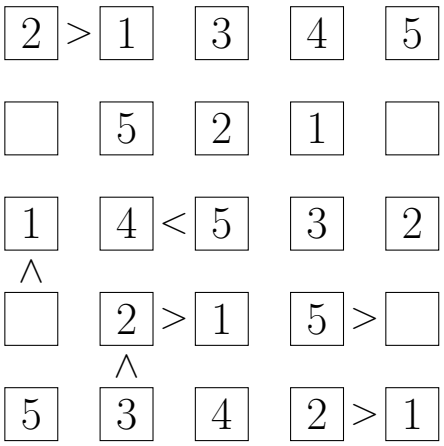


Figure 7: A 5×5 Futoshiki puzzle with only four empty cells that is not uniquely solvable.

The puzzle that was generated in this case cannot be considered interesting to solve. Besides not being challenging, we also see that the inequality signs provide no information. Most inequality signs are between two filled cells, and the remaining ones tell us that a cell must be greater than 1 or less than 5, but this is true for all numbers in the same row or column. Even though numerous inequality signs were added to the puzzle, none are actually useful and we may as well remove all of them. We first address this issue. Considering we want to keep as many inequalities as possible, we remove numbers from cells adjacent to an inequality sign. If both adjacent cells are filled, then one number must be removed. We remove one at random, and if this causes the puzzle to have multiple solutions, we add the number back and remove the other number. If the puzzle still does not retain a unique solution, then we add the number back and resort to removing the inequality sign. Now, any extreme values (1 and $N$) adjacent to an inequality sign must be removed, as this always causes the sign to add no information. We again remove the number, and if this creates multiple solutions, we add it back and remove the inequality sign.

The process of handling uninformative inequality signs is shown in Algorithm 3. After applying this, all inequality signs provide some information to solve the puzzle. However, an unfortunately generated puzzle may still contain far too many hints to provide an interesting cognitive challenge. Therefore, we want to know how to construct a minimal puzzle. A minimal puzzle is one where all hints are strictly necessary to solve the puzzle; the removal of any hint causes the puzzle to have multiple valid solutions. Because we prioritize inequality hints, we start by removing number hints. Considering that removing numbers in some order can produce a different minimal puzzle compared to removing numbers in some other order, we randomize the order of removal to eliminate bias. We visit all numbers in this random order and remove them if doing so retains a uniquely solvable puzzle. After this, we do the same for the inequality signs, again in a random order. Algorithm 4 shows how to turn a Futoshiki puzzle into a minimal puzzle. Doing this may remove too many inequality hints. To prevent this, we simply stop the removal of inequality signs if there are only $N$ signs left.

The complete algorithmic process for bottom-up generation is shown in Algorithm 5. With this, we are able to generate random, uniquely solvable Futoshiki puzzles with an appropriate number of inequality signs. During each step, uniqueness must be verified many times over, underlining the importance of selecting the most efficient solving method. Next, we consider how to generate

**Algorithm 3** Handling uninformative inequality hints.

**Pre:** A valid Futoshiki puzzle is given.
**Post:** All remaining inequality hints provide some information to solve the puzzle.

```
for each inequality sign that gives no information do
    if sign is adjacent to an extreme number then                ▷ Adjacent to 1 or N
        remove extreme number
        if puzzle now has multiple solutions then
            add extreme number back
            remove inequality sign
        end if
    else                                  ▷ Both adjacent cells are assigned a number
        randomly choose which adjacent cell to remove the number from
        if puzzle now has multiple solutions then
            add number back
            remove number from other adjacent cell
            if puzzle still has multiple solutions then
                add number back
                remove inequality sign
            end if
        end if
    end if
end for
```

**Algorithm 4** Minimization of hints for a given Futoshiki puzzle.

**Pre:** Start with a puzzle that has a unique solution.
**Post:** Puzzle has minimal hints.

```
for each nonempty cell in a random order do
    num copy = cell
    cell = empty
    if puzzle now has multiple solutions then
        cell = num copy
    end if
end for
for each inequality sign in a random order do
    sign copy = inequality
    inequality = empty
    if puzzle now has multiple solutions then
        inequality = sign copy
    end if
end for
```

**Algorithm 5** Bottom-up approach to randomly generate a puzzle with minimal hints.

**Pre:** Start with an empty puzzle with no inequalities.
**Post:** Puzzle has a unique solution and a minimal number of hints.

```
number of ineqs = random number in range [N, N^2]
for i from 1 to number of ineqs do
    ineq pos = random inequality position
    place random sign at ineq pos
    if puzzle became unsolvable then
        flip sign at ineq pos
    end if
end for
while puzzle has multiple solutions do
    current cell = random empty cell
    for num from 1 to N in random order do
        current cell = num
        if puzzle is still solvable then
            break
        end if
    end for
end while
handle uninformative inequalities with Algorithm 3
minimize hints with Algorithm 4
```

puzzles from the top down.

## 3.2 Top-down approach

To generate puzzles from the top down, we start with a full Latin square, add inequality hints, and then remove numbers as long as the puzzle stays uniquely solvable. To do this, we first need a Latin square. The Latin square that we start with is the unique solution to a puzzle generated from the top down. All solvers discussed in Section 2 can produce a Latin square simply by solving an empty square with no hints. However, if we use a deterministic solver, then the produced Latin square is always the same, meaning that the solution to any puzzle we generate will be identical. To prevent this problem, we can use a nondeterministic solving method. For example, we can use the backtracking algorithm as specified in Section 2.2, but instead of filling the square from top-left to bottom-right, we choose a random order. Furthermore, when placing a number in a cell, we try numbers in a random order, rather than from 1 up to $N$. In fact, even randomizing only the cell order or only the number order allows us to generate all possible Latin squares; there is no need to randomize both orders.

A problem with this approach is that this nondeterministic backtracking solver can require significant computation time to generate a single Latin square, especially for larger dimensions. Alternatively, we can use a trick to allow the deterministic solvers to produce different Latin squares. We first fill the top row of an empty square with a random permutation of the numbers 1–$N$. We then create a random permutation of the top row by using a Fisher-Yates shuffle [20] to guarantee

that each permutation is equally likely to occur. This creates $N!$ possible permutations for the top row, and thus $N!$ different Latin squares can be generated using a deterministic solver. We can do the same with the numbers from the left column, excluding the cell from the top row which already has a value, to further add $(N-1)!$ possibilities for the left column. This means that any puzzle that we generate can have one of $N! \cdot (N-1)!$ different solutions. This is much fewer than the number of different Latin squares for $N \geq 5$ [3], but more than enough to create a large number of varying puzzles. Not only does this allow us to use efficient deterministic solvers, but these solvers also require less work to find a Latin square considering that $N + N - 1$ numbers are already given. This trick is valid because a Latin square only requires the numbers 1–$N$ to appear in each row or column, but the order in any row or column does not matter. Therefore, given some Latin square, we can always swap any row with any other row, or any column with any other column and the result will still be a valid Latin square. In fact, we can also pre-place numbers in the top row and left column in the same way before we fill in the remaining cells while using a nondeterministic solver, while still allowing it to produce every possible Latin square.

We can use this same principle to generate Latin squares even more efficiently. To do this, we start with a known Latin square, or a simple Latin square that can be generated in polynomial time. An example is a Latin square where the top row contains the numbers 1–$N$ in that order, and each remaining row is the previous row shifted by one position. A $5 \times 5$ example is shown in Figure 8. This always results in a valid Latin square. We can shuffle the rows to obtain $N!$ different permutations, and we can do the same for the columns, resulting in $N!$ more permutations. In total, by shuffling all rows and columns of our constructed Latin square we obtain $N! \cdot N!$ permutations. However, some permutations are equivalent, so the actual number of different Latin squares we are able to produce in this way is smaller. Although we are unable to create all possible Latin squares, this method is by far the fastest option. While the previous methods require exponential time, constructing the simple Latin square requires $\Theta(N^2)$ time, and shuffling the rows and columns requires $\Theta(N)$ time.

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| 2 | 3 | 4 | 5 | 1 |
| 3 | 4 | 5 | 1 | 2 |
| 4 | 5 | 1 | 2 | 3 |
| 5 | 1 | 2 | 3 | 4 |

Figure 8: Simple Latin square that can be constructed in polynomial time for any size. The top row contains the numbers 1 to $N$ in this order, and each successive row is shifted by one position.

Now that we are able to create a large number of different Latin squares, we create one at random and continue the Futoshiki generation process. We first add a random number of inequality signs in the range $[N, N^2]$ and for each of these, we select a random inequality position. In top-down

generation, we do not randomly choose which inequality ($<$, $>$, $\vee$ or $\wedge$) to add. Instead, after selecting a position to place an inequality sign at, we look at the numbers of the Latin square that are adjacent to this position and place the correct sign based on which number is greater. This is why we add the inequality signs before removing any numbers, allowing us to place inequalities without worrying about which one to place, or whether placing them would invalidate our puzzle. This allows us to avoid expensive calls to a solver.

Next, we remove numbers adjacent to inequality signs in the same manner as in the bottom-up approach to make sure that each inequality sign provides additional information about the possible solution. Again, we are careful not to blindly remove numbers, but instead check whether removing a number allows us to retain a unique solution. Now, our puzzle is uniquely solvable with a suitable number of inequality signs, but again, it may feature too many hints to present a solving challenge. As we did for the bottom-up approach, we create a minimal puzzle by first removing number hints in a random order as long as the puzzle remains uniquely solvable, and then doing the same for superfluous inequality hints.

**Algorithm 6** Top-down approach to randomly generate a puzzle with minimal hints.

**Pre:** Start with an empty puzzle with no inequalities.
**Post:** Puzzle has a unique solution and a minimal number of hints.

```
generate a random Latin square
number of ineqs = random number in range [N, N^2]
for i from 1 to number of ineqs do
    ineq pos = random inequality position
    place sign according to Latin square at ineq pos
end for
handle uninformative inequalities with Algorithm 3
minimize hints with Algorithm 4
```

The complete algorithmic process for top-down generation is found in Algorithm 6. Compared to the bottom-up approach, placing inequalities is simpler, and there is no need to add numbers. However, the top-down approach requires us to create a method to generate random Latin squares. Now that we understand both approaches, we discuss the differences between them and why we may want to choose one over the other.

## 3.3 Comparing the two approaches

Firstly, the bottom-up (BU) approach allows us to generate all possible puzzles with every possible solution. While this is also possible for the top-down (TD) approach with a nondeterministic solver, this costs significant computation time in a bad case. Alternatively, there are faster methods at the cost of variety. A benefit of TD is that, by starting from a full Latin square, adding inequality signs requires hardly any computation. With BU, each added inequality sign requires a call to a solver to verify validity. If an added inequality sign causes the puzzle to be unsolvable, then a solver must verify that no solution exists; it must rule out all possible assignments. Concluding that a puzzle has zero solutions is therefore often much more expensive than finding two solutions to conclude that it is not uniquely solvable. Especially considering that inequalities are added to an otherwise empty puzzle, there are many possible assignments and ruling out each one is computationally

expensive, which might cause BU generation to take significantly longer than TD generation in general.

Another advantage for TD is that we are easily able to control the number of hints of our puzzle. Since we start from a valid Latin square, we know that removing a number never invalidates a puzzle, though it may cause it to no longer be uniquely solvable. Therefore, we can simply remove hints until we have the desired number of hints or until the puzzle is minimal. In contrast, for the BU method, we have to add hints. This means that we have to choose a random empty cell to place a number into. However, placing a random number can again invalidate our puzzle, and we may have to try many numbers, and thus conclude infeasibility many times, before we place the right number in the cell. Considering this, it seems likely that generating puzzles top-down is not only faster, but also preferable for puzzle design.

# 4 Experiments for solving and generating puzzles

Now that we are able to generate a large number of puzzles at random, we carry out experiments to compare the efficiency of our solving and generation methods. In this section, three experiments are carried out. Although Futoshiki solving is NP-complete, all methods to solve and generate puzzles discussed in the previous sections require relatively little memory: all methods require $\mathcal{O}(N^4)$ space. Rather than memory space, computation time is the limiting factor for these algorithms. Because of this, the experiments in this section focus on time efficiency.

Firstly, the standard backtracking algorithm is compared to the heuristic backtracking algorithm to find out whether spending extra time to choose the next cell is beneficial. Secondly, the solving methods discussed in Section 2 are compared by the computation time required to find a solution to a variety of different puzzles. Lastly, the bottom-up and top-down generation approaches from Section 3 are compared to measure the time required to generate a minimal puzzle.

## 4.1 States visited during backtracking

Compared to the standard backtracking approach, the heuristic backtracking approach takes significantly longer to choose the next cell because it requires keeping track of the possibilities for each cell to choose the one with the least remaining possibilities. This means that in the worst case, the standard approach actually finds a solution considerably faster. However, the heuristic approach likely finds a solution much faster in the average case. To study whether using the heuristic allows us to find a solution earlier, we compare the number of possibilities tried before reaching a solution. We measure the total number of times a number was placed in a cell, which corresponds to a different puzzle state, before reaching a solution for both algorithms. An advantage of counting states is that the results do not depend on implementation or hardware and are consistent regardless of these factors.

We experiment on three different puzzle sizes. We further distinguish between number hints and inequality hints to discover whether the heuristic function is beneficial in a general case. For each configuration, 100 puzzles are randomly generated with a specified number of inequality hints and number hints. For each of these puzzles, the number of states visited before finding a solution is measured for both the standard backtracking approach (BT) and the heuristic backtracking approach ($\mathrm{BT_H}$), and the average number of states per instance is calculated for each configuration.

To generate the puzzles, we use the top-down generation method, as this allows us to easily control the number of hints. Generating puzzles with a specific number of hints means that the generated puzzles are not necessarily uniquely solvable. This does not detract from the relevance of the experiment, as the solvers are frequently employed during generation to conclude whether a puzzle has a unique solution.

Table 3: Comparison of the number of visited states before finding a solution between the standard and the heuristic backtracking approaches on various puzzle configurations. The average number of visited states across 100 instances is shown as well as the maximum difference between the two algorithms on a single instance.

| **N** | **Numbers** | **Inequalities** | **BT** | $\mathbf{BT_H}$ | $\max(\mathbf{BT} - \mathbf{BT_H})$ | $\max(\mathbf{BT_H} - \mathbf{BT})$ |
|---|---|---|---|---|---|---|
| 5 | 0 | 15 | 1 444 | 94 | 10 586 | 299 |
| 5 | 10 | 5 | 28 | 15 | 93 | 0 |
| 5 | 10 | 15 | 24 | 15 | 72 | 0 |
| 9 | 15 | 75 | 72 969 300 | 84 911 | 2 962 356 700 | 255 264 |
| 9 | 40 | 10 | 1 826 | 49 | 32 042 | −6 |
| 9 | 40 | 75 | 169 | 41 | 1 798 | −1 |
| 12 | 50 | 150 | 332 309 000 | 63 720 | 19 603 343 559 | −1 546 |
| 12 | 75 | 25 | 988 156 | 174 | 51 827 244 | −69 |
| 12 | 75 | 150 | 1 977 | 71 | 40 861 | −38 |

The results of the experiment are found in Table 3. We see that the mean number of states visited by $\mathrm{BT_H}$ is significantly lower than the number of states visited by BT for all configurations. Especially for larger configurations with fewer given numbers, BT visits many millions of states on average, while $\mathrm{BT_H}$ encounters a solution much earlier. For five of the nine configurations, $\max(\mathrm{BT_H} - \mathrm{BT})$ is negative, which means that $\mathrm{BT_H}$ visited fewer states than BT for all 100 instances. In the best case for a $12 \times 12$ puzzle with relatively few number hints, $\mathrm{BT_H}$ visited over 19 billion fewer states than BT. However, $\mathrm{BT_H}$ is not always better, as for some instances $\mathrm{BT_H} - \mathrm{BT}$ was positive, meaning BT found a solution earlier than $\mathrm{BT_H}$ for these instances.

We also see that $\mathrm{BT_H}$ works especially well for configurations with many hints. For example, for configuration $(9, 40, 75)$, $\mathrm{BT_H}$ found a solution after only 41 states on average. Notice that for this puzzle, there are $9^2 = 81$ cells, 40 of which were already assigned a number. This means that 41 cells are empty, thus $\mathrm{BT_H}$ was able to find a solution in the minimum possible number of visited states in the average case. In puzzles with many hints, the possibilities for each cell are more restricted and therefore $\mathrm{BT_H}$ benefits more from choosing the cell with the fewest possibilities. We also see this in the fact that while BT struggled the most with the large puzzle instances, $\mathrm{BT_H}$ actually required relatively fewer states on average for the $12 \times 12$ puzzles than the smaller $9 \times 9$ puzzles. While there are more empty cells to fill, these larger configurations feature more hints and are therefore more restricted.

It seems that in general, number hints are much more restricting than inequality hints. Configuration $(12, 50, 150)$ required the largest number of visited states to find a solution. However, only adding 25 number hints while removing 125 inequality hints (configuration $(12, 75, 25)$) resulted in both methods being able to find a solution much faster.

## 4.2 Time efficiency of puzzle solving methods

In this experiment, we compare the solvers from Section 2 by the average amount of time each solver requires to find a solution to a puzzle. Taking into account the result of the previous experiment, we do not consider BT in this experiment. Instead, we compare the remaining three solvers with the most promise: $\text{BT}_\text{H}$ SAT, and ILP. For the SAT solving method, the `CaDiCaL` solver [21] [22] is used due to its modern performance and convenient `C++` API. The ILP solver used is `SCIP` [23], also for its modern performance. With it, the `C++` wrapper `SCIP++` [24] is used for easy integration into our project. An important note is that the performance of the SAT and ILP solving methods is largely dependent on these solvers. If other solvers were used, the results of this experiment would be different.

The solving methods are tested on puzzles of varying dimensions and number of hints. We are also interested to see whether any solver benefits more from one type of hint compared to the other. Therefore, we first consider instances where the number of inequality signs stays constant for the same puzzle size, while the number of number hints changes. We have chosen to include $N$ inequalities: the minimum number of inequalities that a Futoshiki puzzle should have, as discussed in Section 3. After this, we study the effect of increasing the number of inequality signs while the number of number hints stays constant.

For each configuration, 100 puzzles are generated and the average computation time to find a solution is measured. Considering that all methods are able to find solutions to puzzles with small dimensions in little time, this experiment includes puzzles of larger sizes. To generate these large puzzles efficiently, top-down generation is used, starting from a simple Latin square and shuffling the rows and columns for variety.

The results of the first part of this experiment are found in Table 4. We see that all solvers are able to quickly find a solution to small configurations. Even for the $10 \times 10$ puzzles, which is already larger than any standard puzzle, all solvers are able to find a solution in less than a second on average for all configurations.

Like in the previous experiment, we see that $\text{BT}_\text{H}$ performs the best with the most restricted puzzles. For the largest $25 \times 25$ puzzles, $\text{BT}_\text{H}$ finds a solution almost instantly, while the other two solvers require up to several seconds. However, we also notice that $\text{BT}_\text{H}$ encounters bad cases more frequently: an instance where the solver requires a much larger amount of time to find a solution. For the largest configurations with the least number of hints, $\text{BT}_\text{H}$ encountered instances which it was unable to solve within 10 hours. For all instances it was able to finish, we see that the standard deviation is much higher compared to the other methods. This shows that in most cases the algorithm solved the puzzle in a very short time, but in a few cases it got stuck for a significant amount of time. In fact, $\text{BT}_\text{H}$ had the largest standard deviation in relation to its solving time for all puzzle configurations. While $\text{BT}_\text{H}$ usually finds a solution quickly, the other two algorithms are more stable and thus arguably more reliable.

For smaller puzzles, SAT outperformed ILP. However, all solvers are able to solve small puzzles quickly, and the difference between SAT and ILP is minor. For larger puzzles on the other hand, ILP is significantly faster. This might indicate that ILP has more overhead, which causes it to appear relatively slow on small puzzles. The computation time of both SAT and ILP decreases at roughly the same rate as the number of number hints increases.

To see whether the same pattern occurs when the number of inequalities increases, we consider the two hardest configurations that all solvers were able to complete. We gradually increase the

Table 4: Comparison of the computation time to find a solution to a puzzle between all solving methods on various puzzle sizes. For each size, the number of inequalities stays constant while the number of number hints increases. The average time across 100 instances is measured and shown in seconds, along with the standard deviation. Solvers that got stuck for over 10 hours on a single instance were halted and marked with **D.N.F.** (Did Not Finish) for that configuration.

| **N** | **Numbers** | **Inequalities** | **$BT_H$** | **SAT** | **ILP** |
|---|---|---|---|---|---|
| 7 | 0 | 7 | $< 0.01$ | $0.02 \pm 0.01$ | $0.05 \pm 0.04$ |
| 7 | 10 | 7 | $< 0.01$ | $0.01 \pm 0.00$ | $0.03 \pm 0.01$ |
| 10 | 20 | 10 | $< 0.01$ | $0.34 \pm 0.12$ | $0.44 \pm 0.27$ |
| 10 | 30 | 10 | $< 0.01$ | $0.16 \pm 0.06$ | $0.18 \pm 0.13$ |
| 10 | 40 | 10 | $< 0.01$ | $0.07 \pm 0.03$ | $0.12 \pm 0.04$ |
| 15 | 100 | 15 | **D.N.F.** | $5.89 \pm 2.07$ | $2.80 \pm 1.80$ |
| 15 | 105 | 15 | $41.16 \pm 268.16$ | $3.97 \pm 1.93$ | $2.05 \pm 1.63$ |
| 15 | 110 | 15 | $3.92 \pm 22.68$ | $2.45 \pm 0.96$ | $1.56 \pm 1.28$ |
| 15 | 120 | 15 | $< 0.01$ | $0.87 \pm 0.34$ | $0.59 \pm 0.35$ |
| 15 | 130 | 15 | $< 0.01$ | $0.29 \pm 0.13$ | $0.24 \pm 0.11$ |
| 25 | 400 | 25 | **D.N.F.** | $43.14 \pm 31.75$ | $16.90 \pm 20.08$ |
| 25 | 405 | 25 | $49.95 \pm 196.91$ | $32.98 \pm 29.95$ | $9.66 \pm 12.79$ |
| 25 | 410 | 25 | $5.48 \pm 23.22$ | $18.17 \pm 12.10$ | $5.41 \pm 12.69$ |
| 25 | 420 | 25 | $0.02 \pm 0.10$ | $5.39 \pm 2.57$ | $1.81 \pm 0.91$ |
| 25 | 430 | 25 | $< 0.01$ | $2.62 \pm 1.19$ | $1.20 \pm 0.76$ |

number of inequalities to our upper limit of $N^2$ while keeping the number of number hints constant, measuring the performance at each step.

The results of the second part of this experiment are found in Table 5. We see that only increasing the number of inequality hints has the same general effect as increasing the number of number hints. With many added hints, $BT_H$ becomes able to solve most instances in a fraction of a second. However, just as in the experiment in Section 4.1, we see that a greater number of inequality hints is needed to create the same effect as adding only a few number hints, signifying that number hints provide more information in general across solving methods.

As SAT requires $N \cdot \frac{N-1}{2}$ clauses to encode an inequality while ILP requires only a single linear constraint, one might expect ILP to perform better as the number of inequality signs increases. However, similar to the effect we see when increasing the number of number hints, the computation time of SAT and ILP decrease at roughly the same rate as inequalities are added.

From this experiment it becomes clear that small configurations are most quickly solved by SAT, while large configurations are solved fastest by the ILP method. $BT_H$ requires the least computation time for significantly restricted puzzles of all sizes, but is less reliable, as a bad case requires a long time to solve in comparison to the other methods. To most efficiently solve a puzzle, the best option may be to examine the puzzle and choose which solving method to inquire based on the dimensions and number of hints.

Table 5: Comparison of the computation time to find a solution to a puzzle between all solving methods on two difficult configurations with an increasing number of inequality hints while the number of number hints stays constant. The average time across 100 instances is measured and shown in seconds, along with the standard deviation.

| N | Numbers | Inequalities | $\mathbf{BT_H}$ | SAT | ILP |
|---|---|---|---|---|---|
| 15 | 105 | 50 | $8.11 \pm 52.67$ | $2.30 \pm 0.96$ | $1.60 \pm 1.52$ |
| 15 | 105 | 85 | $1.05 \pm 5.44$ | $1.85 \pm 0.79$ | $1.32 \pm 2.42$ |
| 15 | 105 | 155 | $0.01 \pm 0.03$ | $0.54 \pm 0.30$ | $0.16 \pm 0.11$ |
| 15 | 105 | 225 | $< 0.01$ | $0.16 \pm 0.07$ | $0.08 \pm 0.02$ |
| 25 | 405 | 60 | $5.10 \pm 19.00$ | $21.03 \pm 10.82$ | $4.09 \pm 4.10$ |
| 25 | 405 | 100 | $0.22 \pm 0.87$ | $11.28 \pm 6.57$ | $1.68 \pm 0.76$ |
| 25 | 405 | 175 | $0.04 \pm 0.38$ | $6.06 \pm 3.05$ | $1.33 \pm 0.76$ |
| 25 | 405 | 325 | $< 0.01$ | $2.90 \pm 1.17$ | $0.56 \pm 0.38$ |
| 25 | 405 | 475 | $< 0.01$ | $1.22 \pm 0.60$ | $0.41 \pm 0.12$ |
| 25 | 405 | 625 | $< 0.01$ | $0.60 \pm 0.26$ | $0.37 \pm 0.01$ |

## 4.3 Time efficiency of puzzle generation methods

To find out which puzzle generation strategy is most efficient, we compare the time required to generate 100 minimal puzzles, as defined in Section 3. In Section 4.2, the time each solver required to find a single solution was measured and compared. On the other hand, generating puzzles involves not only finding a single solution, but also concluding that no solution is possible, or that not more than one solution is possible. In these cases, the solvers must rule out all possibilities, rather than finding a single satisfying assignment. Because of this, the method that required the least time to find a solution may not be the best method to generate a puzzle. Therefore, we also compare the solving methods in this experiment. For various puzzle sizes, we measure the time required to generate 100 puzzles using each solver.

In Table 6, we see that the TD generation method on average requires less computation time in all cases. For most configurations, BU required 2–3 times as much time as TD. As discussed in Section 3, this is likely because adding inequality hints is much more computationally expensive and verifying a near-empty puzzle requires checking many more possibilities on average than verifying a near-full puzzle.

Interestingly, while ILP overall performed best in the previous experiment, here it performs relatively poorly. This may be because the puzzles generated are of small size, and while the other methods were only slightly faster than ILP on small puzzles, puzzle generation requires repeated solving and verification. If, for each inquiry, ILP is only slightly slower than the other methods, this small difference quickly adds up the more the solver is used. The difference between ILP and the other methods could also mean that ILP is effective at finding a single satisfiable assignment, but is less efficient at concluding that a set of linear constraints is infeasible, which is frequently required to verify that a puzzle has a unique solution.

We also see that $\mathrm{BT_H}$ is the fastest option for generating small puzzles, but, as in the previous experiment, it struggles when the number of possibilities increases, as this raises the chance of making an unfavorable decision. For these reasons, SAT in general seems to be most effective for

Table 6: Comparison of the computation time required by the two generation methods to generate minimal puzzles of different sizes using each solver. For each size, approach, and solving method, 100 puzzles are generated. The average time in seconds to generate a puzzle is shown.

| N | Solver | BU | TD |
|---|---|---|---|
| 4 | $BT_H$ | $< 0.01$ | $< 0.01$ |
| | SAT | 0.09 | 0.03 |
| | ILP | 0.30 | 0.15 |
| 5 | $BT_H$ | $< 0.01$ | $< 0.01$ |
| | SAT | 0.51 | 0.20 |
| | ILP | 1.06 | 0.51 |
| 6 | $BT_H$ | 0.11 | 0.04 |
| | SAT | 1.56 | 0.56 |
| | ILP | 5.17 | 2.58 |
| 7 | $BT_H$ | 19.96 | 9.71 |
| | SAT | 9.71 | 4.71 |
| | ILP | 34.51 | 31.28 |

puzzle generation purposes. However, the best approach might again be to combine these methods, choosing $BT_H$ for small puzzles and SAT for larger ones.

Even with the fastest options, generating a single minimal $7\times7$ puzzle requires several seconds. However, in practice, it is often not necessary to generate a minimal puzzle, as the small number of hints often makes these puzzles incredibly difficult to solve for a human. To devise a method for generating puzzles of appropriate difficulty, we first consider how to determine the difficulty of a Futoshiki puzzle.

# 5 Difficulty of human Futoshiki solving

Some Futoshiki puzzles are more difficult for humans to solve than others. To design puzzles with an appropriate difficulty level, we need to understand which puzzles are more difficult to solve than others and what it is that makes solving this puzzle more complicated. In the context of puzzle design and solving, difficulty can be considered highly subjective as it depends on the strategy of the solver and the choices made while solving the puzzle. For example, the solving methods presented in Section 2 have different approaches to solve a puzzle. If we were to use any of these methods as an indication for the difficulty of a puzzle, for example by using the computation time to solve the puzzle, the rating will be biased in some way. The backtracking methods, for example, attempt to fill in some cell with a number from 1 to $N$ in that arbitrary order, continuing on to the next cell while the puzzle is still valid until we find a solution. This means that if the very first cell we try to fill should contain the number 1, we would find a solution in less time than for a similar puzzle for which the first empty cell should contain the number $N$. If we were to use the computation time of this solving method to determine the difficulty, puzzles with a solution that contains smaller numbers early on would be classified as less difficult than puzzles with a solution that contains larger numbers early on, even though these puzzles are fundamentally the

same. If we choose the SAT solving method, we would rely on the specific SAT solver that we choose for solving the CNF formula. Each SAT solver has its own design choices, so a solver may find a solution to some puzzles much quicker than some other solver, while being slower than this solver on other puzzles. For solving purposes, we want to use a SAT solver that is generally efficient at finding a solution and is easily integrated into our program, while caring less about the specific implementation details. Considering we want to rate the difficulty of a puzzle not only accurately but also consistently, using our SAT solving method is not an option. Furthermore, if a revolutionary new SAT solver is developed we may want to use its solving capabilities and replace our current solver without having to worry about accidentally changing the difficulty rating system of the puzzles. For the same reasons, the ILP solving method is also unfit for assigning difficulty. Even if we created a computer program that solves Futoshiki puzzles without any bias regarding puzzle configuration, using the computation time of this program as a difficulty measure only indicates how difficult it is for the computer program to solve, and may well have little relevance to how difficult solving the puzzle is for a human solver. After all, any somewhat efficient computer program is able to solve all puzzles one finds in a puzzle book in under a second, while we often require a few minutes at best. We need to find some other metric to determine difficulty that is relevant to human puzzle solvers.

On a surface level, an unsolved Futoshiki puzzle has two properties: the puzzle size and the number of hints. Futoshiki puzzles usually range from size $4\times4$ to $9\times9$. The hints are placed to enforce a unique solution and to allow us to make logical inferences to complete the rest of the puzzle. Hints for a Futoshiki puzzle can further be divided into two types: number hints and inequality hints. In general, we consider larger puzzles more difficult, as cells have more possibilities and more cells are involved in making logical inferences. For example, any cell in a $9\times9$ puzzle directly affects the numbers in the 16 other cells in its row and column, and indirectly affects all other 64 cells in all the other rows and columns. On the other hand, a cell in a $4\times4$ puzzle only directly affects 6 other cells and indirectly affects 9 cells. Solving a $9\times9$ puzzle thus requires much more cognitive load in general compared to puzzles with smaller dimensions. We also say that in general, a puzzle with more given hints is more easily solved than a puzzle with fewer hints. The more hints available, the more information the solver has access to, and the fewer empty cells remain to be filled. As we observed in Section 4, number hints provide us with more information than inequality hints. A number hint directly restricts all cells in its row and column, while an inequality hint only directly restricts the two cells adjacent to it. Therefore, we say that in general, a puzzle with more number hints compared to inequality hints is easier to solve than a puzzle with the same number of total hints, but with a greater proportion of inequality hints. Are these properties alone enough to accurately determine a puzzle's difficulty?

Figure 9 shows two puzzles. Both puzzles are $4\times4$ puzzles, have two number hints, four inequality hints, and a unique solution. Despite this, most humans will find that the puzzle on the left is significantly easier to solve than the puzzle on the right. This shows that the puzzle dimensions and the number of hints do not provide us with enough information to determine puzzle difficulty — the placement of the hints also plays a role. To gauge how difficult a puzzle is to solve for a human solver, we study how humans approach solving these puzzles.

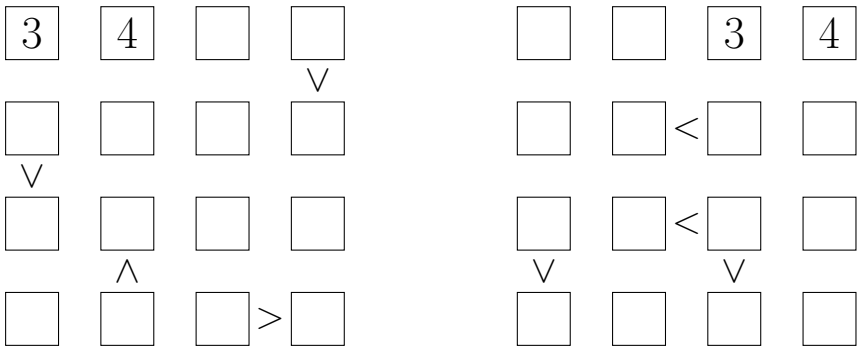


Figure 9: Two 4×4 Futoshiki puzzles. With the same number of hints, the puzzle on the right is generally more difficult to solve than the puzzle on the left.

## 5.1 Candidate list strategy

The candidate list strategy is a strategy that human solvers often use for similar Latin square completion puzzles such as Sudoku [25]. This strategy involves keeping track of all the candidates for each cell in the puzzle and eliminating candidates using logical inference techniques until the puzzle is solved. Initially, all cell candidates are written down according to the direct possibilities: a number is marked down in a cell if this number does not appear in its row or column and, in the case of Futoshiki puzzles, if it is in accordance with adjacent inequalities.

To further reduce the candidates, consider two cells $A$ and $B$ on an $N \times N$ puzzle where an inequality sign specifies that $A > B$. In this case, even if both cells are empty, we know that cell $A$ can never be assigned the number 1 since cell $B$ must then be assigned a number smaller than 1 which is not possible. For the same reason, the number $N$ cannot be placed in cell $B$ as there is no number greater than $N$. We remove both of these numbers as candidates in their respective cell. Maintaining a full list of candidates for each cell helps us identify which inference techniques can be used to solve the puzzle. However, even without explicitly maintaining a (full) list of candidates, these techniques can be used to solve a Futoshiki puzzle.

By understanding these techniques, we can write a program to simulate human Futoshiki solving, revealing which solving techniques are needed to complete a puzzle. We can use this information to rate the difficulty of a puzzle: if only basic techniques are sufficient, then we know that the puzzle can be solved even by inexperienced puzzle solvers and that this puzzle might be uninteresting to avid puzzlers. However, if basic techniques are insufficient and more advanced techniques are required, then only Futoshiki experts might be able to solve this puzzle. Many of the techniques used to solve Sudoku puzzles are inferences based on the Latin square constraint, which means that we can also use these techniques to solve a Futoshiki puzzle. In the following section, these techniques are explained, with their names taken from [25], and two techniques are added that use the inequality signs to eliminate candidates. The techniques are roughly ordered from basic to advanced.

### 5.1.1 Naked single

A naked single is a cell that only has one candidate. In this case, we can place this candidate in the cell as no other numbers are valid.

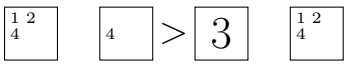


Figure 10: Example of an isolated row of a 4×4 Futoshiki puzzle where a naked single is present in the second cell.

An example of a naked single in an isolated row is shown in Figure 10. The second cell must contain a number greater than the number in the third cell, which holds the number 3. For $N = 4$, the number 4 is the only candidate. Observing this naked single, we place the number 4 in the second cell, and remove 4 as a candidate from all other cells in its row and column.

### 5.1.2 Hidden single

A naked single is a cell with only one possible number. In contrast, a hidden single is a number with only one possible cell in a row or column. All other cells in the row or column do not have this number as a candidate. An example of a hidden single is found in Figure 11. Each cell has at least two possibilities, so there is no naked single. However, in the top row, the number 1 cannot be placed in the first, second, or fourth column. Therefore, we know that we must place the number 1 in the cell in the third column of the top row. Similarly, a hidden single is found in cell $(c, 4)$, as this is the only cell in column 4 where a 1 can be placed.

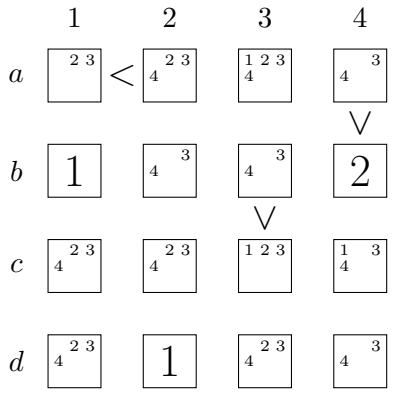


Figure 11: Example of a 4×4 puzzle with a hidden single in cells $(a, 3)$ and $(c, 4)$.

The naked single and hidden single techniques allow us to directly place a number in a cell. With the remaining techniques, this is not possible. Instead, these techniques allow us to eliminate candidates from cells. In case the naked single and hidden single techniques cannot be used, repeated use of the following techniques allows us to remove enough candidates to reveal more naked or hidden singles, eventually allowing us to complete the entire puzzle.

### 5.1.3 Inequality inference

This technique uses the inequality signs, making it our first technique that is specific to Futoshiki puzzles. The technique is outlined in [7] and is used to measure how frequently inequality constraints were used to solve a puzzle in relation to the Latin square constraint.

Consider cells $A$ and $B$, with $A < B$. If the number 1 is eliminated as a candidate for cell $A$, then cell $A$ must receive a number greater than 1: $A \geq 2$. Considering that 2 is the smallest value that $A$ may receive, and $B > A$, we know that in any case, $B > 2$. We can eliminate number 2 as a candidate in cell $B$.

More generally, if $A < B$, then $B > \min(candidates(A))$. If $k$ is the smallest number among all candidates of $A$, then we remove from $B$ all candidates $\leq k$. Of course, the opposite is also true. If $A < B$ and $k$ is the largest number among all candidates of $B$, then we eliminate from $A$ all candidates $\geq k$. This exact situation occurs in Figure 11 in cells $(a, 1)$ and $(a, 2)$. The smallest candidate from cell $(a, 1)$ is the number 2. Since cell $(a, 2) >$ cell $(a, 1)$, cell $(a, 2) > 2$. We can remove 2 as a candidate in cell $(a, 2)$.

With just these first three techniques, the puzzle in Figure 11 can be solved. Notice that repeatedly applying this technique also covers inequality chains: a chain of multiple cells with inequality signs between them. The puzzles in Figure 6 provide an example: despite featuring a small number of hints, they can also be solved solely using these first three techniques.

### 5.1.4 Naked pair

A naked pair occurs when a row or column has two cells with the same two candidates as its only candidates. Say that cell $A$ and cell $B$ are in the same row and have 1 and 2 as the only candidates. Cell $A$ will be assigned one of the two options and the remaining option is assigned to cell $B$. In either case, we know that 1 and 2 cannot be assigned to any other cell in this row. Therefore, we can remove them as candidates for the other cells in the row.

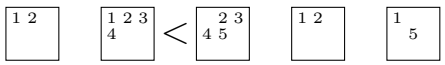


Figure 12: Example of an isolated row of a $5 \times 5$ puzzle where we can use the naked pair technique to eliminate options in the second, third, and fifth cell.

This situation is visualized in Figure 12. In the first and fourth cell, 1 and 2 are the only options. We know that these numbers cannot appear anywhere else in the row, so we remove them from the second, third and fifth cell. This leaves a single candidate for the fifth cell, allowing us to use the Naked Single technique to place a 5 there. This in turn leaves the second and third cells with candidates 3 and 4 — we use the inequality hint to place 3 in the second cell and 4 in the third cell.

### 5.1.5 Hidden pair

The hidden pair technique can be used when there are two numbers which can only be placed in the same two cells in a row or column and are invalid for any other cell. If numbers 3 and 4 are candidates for cells $A$ and $B$ in the same row but for no other cells in that same row, we know that cells $A$ and $B$ must be assigned numbers 3 and 4. Thus, we can remove all other candidates from these cells.

An example is shown in Figure 13. We see that the numbers 3 and 4 are both candidates for the first and second cells, but for no other cells in the row. That means that these numbers must be placed in these cells, and we can remove all other candidates from the first and second cell. With the help of the inequality sign, we place 4 in the first cell and 3 in the second cell.

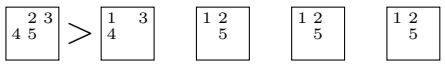

Figure 13: Example of an isolated row of a 5×5 Futoshiki puzzle where the hidden pair technique can be used on the first and second cell.

### 5.1.6 Double inequality inference

We can use this technique when there is a cell with the same inequality to two different cells in the same row or column. Say that cells $A$, $B$ and $C$ are in the same row and $A < B > C$. In this situation, we can use this technique to eliminate candidates from $B$. With the single inequality inference technique, we say that $B > \min(candidates(A))$ and $B > \min(candidates(C))$. Imagine that $candidates(A) = \{2, 4\}$ and $candidates(C) = \{2, 5\}$. In this case, we would say that $B > 2$. However, we know that $A$ and $C$ cannot both be assigned the number 2, as they appear in the same row or column. If $A = 2$, then $C = 5$ and $B > 5$, but if $C = 2$, then $A = 4$ and $B > 4$. In either case, we can eliminate candidates from cell $B$. In general, if $A < B > C$ in the same row or column, then the number in cell $B$ is greater than the second minimum value of the set of candidates of cells $A$ and $C$ combined. In our example, $candidates(A, C) = \{2, 4, 5\}$, and so $B > 4$, the second smallest value. If, on the other hand, $A > B < C$, then we know that $B$ should be less than the second largest value of the set of candidates of cells $A$ and $C$, and we eliminate all candidates from cell $B$ that are greater than or equal to this value.

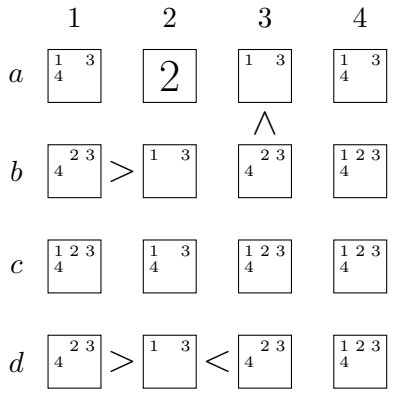

Figure 14: Example of a 5×5 Futoshiki puzzle where the double inequality inference technique can be used to eliminate the number 3 as a candidate from cell $(d, 2)$.

An example is shown in Figure 14. Looking at square $(d, 2)$, we see that its horizontal neighbors should both have a greater value. The second maximum of its neighbors is 3, so cell $(d, 2) < 3$; we eliminate 3 as a candidate. This allows us to use naked singles to fill in the entire second column, and we are able to solve the puzzle with all techniques up to this point.

### 5.1.7 Naked and hidden multiples

A naked triple is like a naked pair, but instead of two cells having the same two candidates, there are three cells in the same row or column that have the same three candidates. One difference is that these three cells do not have to have all three numbers as a candidate; having a subset of

them also suffices. For example, three cells in the same row all with candidates $\{1, 2, 3\}$ qualify as a naked triple, but so do three cells where the first cell has candidates $\{1, 2\}$, the second has candidates $\{1, 3\}$, and the third has candidates $\{2, 3\}$. In both cases, we can remove the numbers 1, 2 and 3 from all other cells in the same row. Similarly, we can identify hidden triples, naked or hidden quadruples, quintuples, and so on, provided that the dimensions of the puzzle permit them. Because more cells are involved and because this technique can involve combining different subsets of candidates, naked and hidden multiples are distinguished from naked and hidden pairs as a separate and more advanced technique as they may be harder to recognize.

We do not necessarily have to distinguish between naked multiples and hidden multiples. This is because if there are $M$ empty squares in a row, then finding a naked multiple of size $K$ is equivalent to finding a hidden multiple of size $M - K$. If we consider the example of an empty row of a $5\times5$ puzzle, where three cells all have candidates $\{1, 2, 3\}$, and the other two cells have all numbers from 1 to 5 as candidates, then we can remove candidates 1, 2 and 3 from the last two cells by recognizing the first three cells as a naked triple. Similarly, we can recognize that the last two cells form a hidden pair with the candidates 4 and 5, and remove all other candidates from these cells. Both techniques lead to the exact same result. In fact, we can recognize this in the naked single example in Figure 10, where besides the naked single, the other two cells form a hidden pair. The same is true for the other examples: in Figure 11 we find a naked triple or naked pair for both hidden singles, and in Figure 12 and Figure 13, where we find a hidden triple and naked triple respectively. From this observation, we say that naked multiples of size $K$ with $K > 2$ are never required to solve puzzles of size $N \leq 5$. If a naked triple was needed to solve the puzzle, then we could always use a hidden pair instead for $N = 5$. In general, puzzles of size $N$ only require naked or hidden multiples of size $\lfloor \frac{N}{2} \rfloor$ in the worst case.

To algorithmically find a naked multiple, we need to find a set of empty cells in a row or column where the number of cells in our set is smaller than the total number of empty cells in the row or column. Within our set of cells, the size of the combined set of candidates must be exactly as large as the number of cells in our set. In other words, we must find a subset of cells of size $M$ where all candidates in the cells amount to $M$ unique candidates across the cells, regardless of how these candidates are distributed in each of the cells. We can solve this problem by looping over all possible sets of candidates of size $> 2$, considering that a set of size 1 is a naked single and a set of size 2 is a naked pair. For each possible set, we count the number of cells for which its set of candidate is fully contained in our set. If the number of cells is equal to the size of our set, then we have found a naked multiple and we check if any candidates from other cells in the same row or column can be eliminated. Although all other techniques up to this point can be computed in polynomial time, for this technique, each possible set of candidates of size $> 2$ must be considered in the worst case. Considering that each number can be included or excluded from the set, there are $2^N$ possible sets, thus requiring exponential time to compute.

We established that a naked multiple of size $K$ in a row or column with $M$ empty cells is equivalent to a hidden multiple of size $M - K$. Therefore, any hidden multiple is found by searching for its corresponding naked multiple; there is no need to create separate methods for algorithmically finding hidden multiples.

### 5.1.8 X-Wing

An X-Wing occurs when there are two rows where some number is a candidate in the same two columns, and in no other cells in that row. Of course, two columns where some number is a candidate in the same two rows is also an X-Wing. For example, rows $b$ and $c$ both have the number 1 as a candidate in the leftmost two columns, so cells $(b, 1)$, $(b, 2)$, $(c, 1)$ and $(c, 2)$ all have the number 1 as a candidate, but no other cells $(a, x)$ or $(b, x)$ with $x > 2$ have 1 as a candidate. In this case, we know that we will either assign 1 to cells $(b, 1)$ and $(c, 2)$, or to cells $(b, 2)$ and $(c, 1)$. In both cases, a 1 will be placed in columns 1 and 2 in either row $b$ or $c$. Therefore, the number 1 cannot be assigned to any other cell in columns 1 or 2, and can be eliminated as a candidate from these cells.

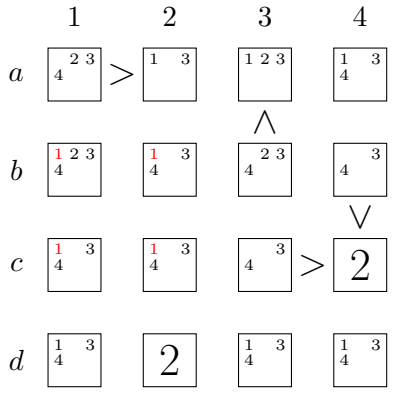


Figure 15: Example of a 4×4 puzzle where the X-Wing technique can be used for the number 1 for rows $b$ and $c$ in columns 1 and 2.

This example is shown in Figure 15. We can remove the number 1 as a candidate from cells $(a, 2)$ and $(d, 1)$. Now the puzzle can be solved entirely by finding naked singles. In general, if a number $k$ is a candidate of the same two columns in two distinct rows, and no other columns in these rows, then $k$ cannot be a candidate of any other cells in these columns. The same applies if we exchange rows and columns.

### 5.1.9 XY-Wing

Imagine that cell $(a, 1)$ has only two candidates $\{1, 2\}$. Suppose there is another cell in row $a$, say cell $(a, 4)$, with $\{1, 3\}$ as candidates. Now, if there is some cell in column 1, say $(c, 1)$, with candidates $\{2, 3\}$, then we have found an XY-Wing. In this scenario, if we assign 1 to cell $(a, 1)$, then cell $(a, 4)$ must be a 3. On the other hand, if we assign 2 to cell $(a, 1)$, then cell $(c, 1)$ must be a 3. In either case, there will be a 3 either in column 4 or in row $c$, therefore, we know that cell $(c, 4)$ cannot be a 3 and we can eliminate it as a candidate for this cell.

In general, an XY-Wing is a situation where there is a cell $A$ with two candidates $\{x, y\}$, a second cell in the same row in column $j'$ with candidates $\{x, z\}$ and another cell in the same column as cell $A$ but in row $i'$ with candidates $\{y, z\}$, then we can remove $z$ as a candidate from cell $(i', j')$.

In Figure 16 we can see such a situation. From cell $(d, 3)$, we can use cells $(a, 3)$ and $(d, 4)$ to eliminate 1 as a candidate from cell $(a, 4)$. After this, the puzzle can be solved with naked singles.

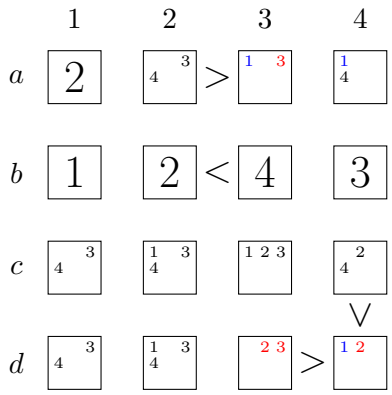


Figure 16: Example of a $4 \times 4$ Futoshiki puzzle with an XY-Wing from cell $(d, 3)$ to eliminate the number 1 as a candidate from cell $(a, 4)$.

### 5.1.10 Swordfish, jellyfish, squirmbag

A swordfish is an X-Wing that concerns three rows or columns rather than two. This means that three rows have some number as a candidate in the same three columns, and we can eliminate this number as a candidate from all other cells in those columns. Similarly to the naked multiple, these three rows do not need to have all three columns as a candidate for this number; a subset of these cells also qualifies. This means that algorithmically finding an X-Wing of arbitrary size also requires exponential computation time.

A jellyfish is an X-Wing that concerns four rows. On a puzzle of size $N$, just like a naked multiple, it is never necessary to find an X-Wing of size $\lfloor \frac{N}{2} \rfloor + 1$ or greater, as this implies that there is an X-Wing of smaller size among the remaining cells. Because of this, the author of [25] states that an X-Wing of size 5, which is called a squirmbag, is never needed to solve a $9 \times 9$ Sudoku puzzle. For Futoshiki puzzles of size $N \geq 10$ on the other hand, we may require the squirmbag to solve the puzzle, though puzzles of this size are not commonly featured in puzzle books.

### 5.1.11 Trial-and-error/guessing

With all techniques described up to this point, we are able to solve many Futoshiki puzzles of various difficulties. However, if we are still not able to solve a certain puzzle, we may need to resort to trial-and-error. We find the cell with the least number of candidates, place one of the candidates in the cell at random, and continue solving the puzzle. If at a later moment we encounter a contradiction, for example, when two cells in the same row or column are forced to have the same number, then we know that the number we guessed was incorrect. At this point, we erase all progress we made after guessing, and remove the number we guessed as a candidate from its cell. If we guess a number and are able to find a solution to the puzzle, then we know that the guess was correct, assuming that the puzzle is uniquely solvable. In theory, any puzzle can be solved just by using trial-and-error. In fact, this is precisely how the heuristic backtracking method from Section 2.3 solves puzzles. Performing this process manually can be quite tedious and requires little logical reasoning. Furthermore, it possibly takes a large number of steps before encountering a contradiction, at which point we must return to the exact puzzle state we were in when making the guess. For these reasons, this technique is considered the most advanced.

## 5.2 Assigning a difficulty rating

Now that we understand how to manually solve a Futoshiki puzzle using the candidate list strategy, how can we use this to assign a difficulty rating to a puzzle?

By creating a program that uses these techniques to solve a puzzle, we can determine which techniques are required to solve it. To do this, we want to use the most basic techniques whenever possible and only try a more advanced technique if we get stuck. Algorithmically, if we have an ordering of the techniques, we start by applying the most basic one. If we cannot eliminate any candidates or place any number, then we try the next technique. If a technique allows us to reduce the candidate list, then we return to the most basic technique. Repeating this process until the puzzle is solved allows us to keep track of the most advanced technique required to solve the puzzle and how many times each technique was used. We use this information to assign a difficulty rating. We rate each puzzle from 1 to 10 inclusive. A puzzle of difficulty 1 should be fairly straightforward to solve even for inexperienced puzzle solvers, while a puzzle of difficulty 10 should require significant effort and experience to find a solution.

Firstly, we say that requiring an advanced technique even once makes the puzzle difficult to solve, because someone unfamiliar with this technique will never be able to solve the puzzle. For this reason, we divide the techniques into three categories: easy, medium, and hard. Each category corresponds to a rating range.

Table 7: Distribution of candidate list strategy techniques in terms of difficulty and rating range.

| **Difficulty** | **Technique** | **Rating range** |
|---|---|---|
| Easy | Naked single<br>Hidden single<br>Single inequality | 1–3 |
| Medium | Naked pair<br>Hidden pair<br>Double inequality<br>Naked multiple | 4–6 |
| Hard | X-Wing<br>XY-Wing<br>Swordfish etc.<br>Trial-and-error | 7–10 |

In Table 7 we show how we divide the techniques in terms of difficulty. Each technique that only involves candidates from a single cell or only involves a single number in a row or column is considered easy. If a puzzle can be solved using only these techniques, the puzzle is assigned a difficulty in the range $[1, 3]$. Techniques that involve candidates of multiple cells in a single row or column are considered medium difficulty, and puzzles where the most advanced strategy required falls in the medium category are assigned a difficulty in the range $[4, 6]$. Lastly, techniques that involve candidates of multiple cells across multiple rows and/or columns are considered hard. The puzzles that require these techniques to be solved are rated in the range $[7, 10]$.

We still need to distinguish between, for example, a level 7 and a level 10 puzzle. To do this, we consider the easiest and hardest puzzles possible within a category. The easiest puzzle in the

*hard* category for example, is a puzzle where the least advanced technique in this category, which is the X-Wing, is required only once, and the rest of the puzzle can be completed using only the most basic technique, which is the naked single. This is the case for the puzzle in Figure 15. The hardest puzzle however, theoretically consists of using the most advanced technique at every step. In practice, this is not possible, as less advanced techniques will always be applicable at some point. Furthermore, a puzzle that can only be solved through trial-and-error may be considered too tedious or difficult to provide a stimulating solving experience. Therefore, we define a distribution of the number of times each technique is used that we argue should be rated as the highest difficulty of a certain category. For example, we could consider a puzzle that requires using every technique at least once to be our baseline level 10 puzzle. There are many puzzles where all of the discussed techniques can be used. This is only possible for puzzles of size $N > 5$, because, as mentioned before, naked multiples and swordfishes cannot occur in puzzles smaller than this size. An example of a $6\times6$ puzzle with no given numbers that requires all techniques to solve is shown in Figure 17.

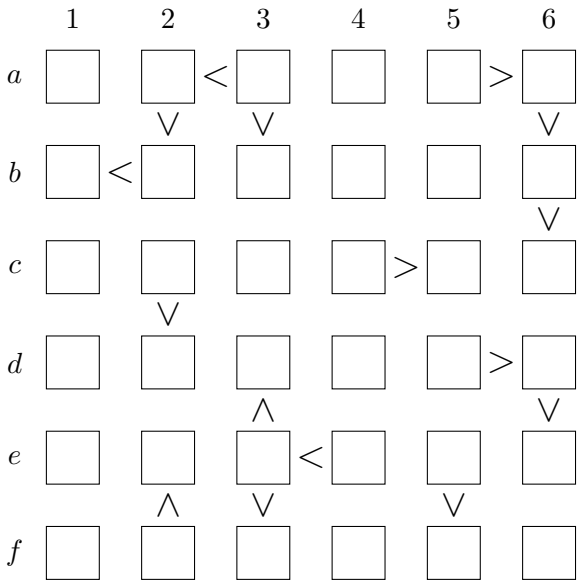


Figure 17: A $6\times6$ puzzle that, when prioritizing basic techniques, requires all techniques to be solved.

Considering that our algorithm to determine which techniques are required for solving a puzzle prioritizes using the least advanced techniques, the most basic techniques are often used the most. Taking this into account, we count how many times any technique was used to find a solution, and from this number, we create a distribution among techniques that should be considered the most difficult level in its category. To generate a decreasing distribution to encode the bias towards basic techniques, we define for each technique $i$ ordered from basic to advanced:

$$distribution[i] = \frac{1}{i^s},$$

where $s$ is a constant: a higher value of $s$ result in a stronger bias towards basic techniques. We create a list of distributions for each technique from the naked single technique up to the most advanced technique in the relevant category. We divide the total number of times any technique

was used among the relevant techniques according to the distribution, while taking care of rounding errors.

For example, consider a puzzle that requires 60 applications of various techniques to solve, and the most difficult technique required is the hidden pair technique. This puzzle is categorized as medium difficulty, and the most difficult technique in this category is the naked multiple. The baseline for a level 4 puzzle is a distribution where a naked pair is used once, and naked singles are used the remaining 59 times. In Table 8 we see an example of what we would consider a baseline level 6 puzzle for the specified number of techniques used with $s = 1.5$. Every technique of its category is used multiple times, with a bias towards more basic techniques.

Table 8: Example of a baseline level 6 puzzle for a puzzle that requires 60 applications of techniques with $s = 1.5$.

| | N. single | H. single | Single ineq. | N. pair | H. pair | Double ineq. | N. multiple |
|---|---|---|---|---|---|---|---|
| Distribution | 1.0 | 0.3536 | 0.1925 | 0.1250 | 0.0894 | 0.0680 | 0.0540 |
| Times used | 32 | 11 | 6 | 4 | 3 | 2 | 2 |

As established in this section, in general larger puzzles are significantly more difficult to solve than smaller puzzles. Larger puzzles have more cells, more candidates per cell, and using a technique involves more cells. To represent this difference in our rating system, we define the value for $s$ in relation to the puzzle size. Considering only the standard puzzle sizes, where the largest puzzles are $9\times9$ puzzles, we define $s = 1.5 - 0.15 \cdot (9 - N)$. With this, larger puzzles are more biased towards the more basic techniques, meaning that even using the more advanced techniques only once may classify it as the most difficult level. On the other hand, smaller puzzles where the most advanced techniques are used just once are rated as a lower level, and level 10 puzzles of this size require using these techniques several times. This helps balance the rating slightly, however, larger puzzles are often still much more difficult to solve compared to smaller puzzles of the same rating.

Having established baselines for the easiest and hardest levels of a category, we use these baselines to assign a specific difficulty level to our puzzle. To do this, we assign a score to our puzzle solution and to both baselines based on how many times each technique was used. For this, we define a weight for each technique in terms of difficulty. A straightforward approach is to assign each technique twice the weight of the previous technique.

The resulting weight distribution is shown in Table 9. For example, using a swordfish is considered equivalent in terms of difficulty to using an X-Wing four times. With these weights, we can multiply the number of times each technique was used by its weight to assign a score. Consider again the example of a medium puzzle with 60 techniques used. The baseline score for a level 4 puzzle would use a naked pair once and a naked single 59 times: the score becomes $1 \cdot 8 + 59 \cdot 1 = 67$. The baseline for a level 6 puzzle according to Table 8 would be $32 \cdot 1 + 11 \cdot 2 + 6 \cdot 4 + 4 \cdot 8 + 3 \cdot 16 + 2 \cdot 32 + 2 \cdot 64 = 350$. Now, a puzzle solution that uses 60 techniques is assigned a score and compared to these baseline values. If the score of our puzzle exceeds 350, it is assigned level 6. If it exceeds the middle value of $\frac{67+350}{2} = 208.5$, then it is assigned level 5. If it is below this value, it is assigned level 4.

Interestingly, with our rating system, we see that the number of hints is often not a good indication of the difficulty of a puzzle. For example, the puzzles in Figure 6 are both rated as level 1 puzzles despite the small number of hints. On the other hand, the puzzle in Figure 18 has many hints but is rated as a level 10 puzzle, requiring the use of the X-Wing technique and multiple uses

Table 9: The weight of each technique in terms of difficulty for assigning a score for our puzzle.

| **Technique** | **Weight** |
|---|---|
| Naked single | 1 |
| Hidden single | 2 |
| Single inequality | 4 |
| Naked pair | 8 |
| Hidden pair | 16 |
| Double inequality | 32 |
| Naked multiple | 64 |
| X-Wing | 128 |
| XY-Wing | 256 |
| Swordfish etc. | 512 |
| Trial-and-error | 1 024 |

of the trial-and-error technique to solve.

One issue with our method is that, when prioritizing basic techniques, the list of required techniques is potentially inaccurate. While this process always finds the most advanced technique required, imagine a puzzle state where no easy or medium techniques can be used. Next, we try using an X-Wing and are able to eliminate a candidate. Then, we try all techniques again but none allow us to make progress. Finally, we use trial-and-error to eliminate a candidate, and after that, we are able to solve the puzzle with only easy techniques. In this case, it is possible that the candidate removed from the X-Wing technique did not help us to solve the puzzle, and we could have solved the puzzle using only the more advanced trial-and-error technique in combination with the easy techniques. The same applies if a technique was used multiple times: perhaps only the last use was needed to make progress. In these cases, the puzzle might be assigned a higher difficulty than desired, as advanced techniques are applied that are not necessary to solve the puzzle. However, it can be argued that humans solve puzzles in a similar way, and that humans rarely use the minimal required techniques to solve a puzzle. Therefore, we do not necessarily see this as a problem.

Another issue is that our method does not distinguish between, for example, a puzzle in which each technique is used once and a puzzle in which each technique is used three times. These puzzles are considered precisely as difficult, as we compute a rating based on the ratio of basic to advanced techniques applied. However, a puzzle that has the same ratio as another puzzle but requires using techniques more often should clearly be considered more difficult.

## 5.3 Generating a puzzle of a specific difficulty

Now that we have a method to assign a difficulty to a puzzle, we consider how to generate puzzles of a certain difficulty. For this, we use the top-down approach as described in Section 3.2, as this allows us to generate puzzles more efficiently and also allows us to more easily control the number of hints. Say we want to generate a puzzle of level $x$. We start with a random Latin square, introduce a random number of inequality signs in our desired range, and begin erasing numbers. Each time a number is erased, we estimate the difficulty as discussed in Section 5.2. If, after erasing a number,

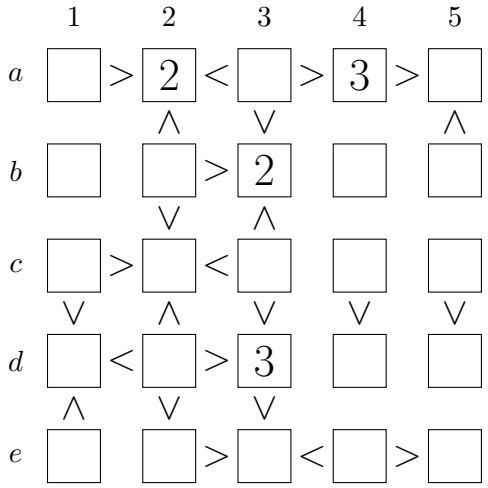


Figure 18: A 5×5 Futoshiki puzzle that, in spite of the large number of hints, is rated as a level 10 puzzle.

our puzzle becomes more difficult than level $x$, we place the number back where it was. If needed, we do the same to minimize the number of inequality signs. Now, when should we stop removing numbers or inequality signs? Intuitively, we might choose to stop when our puzzle is first estimated to be level $x$. However, generating a level 1 puzzle then creates a puzzle in which only a single cell is empty. This puzzle requires one use of the naked single technique to be solved, thus is judged to be level 1. Of course, this puzzle is not interesting to solve, so we want to keep removing numbers. Furthermore, a puzzle with many hints often features several useless hints. An example is shown in Figure 19. We immediately see that the second cell in the top row should be assigned the number 1. This causes the remaining inequality signs to provide no additional information; they may as well have been removed.

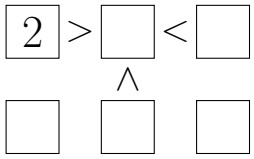


Figure 19: Example of part of a Futoshiki puzzle featuring several useless inequality hints.

One approach we consider to solve these problems is to create a minimal puzzle of a certain difficulty. This means that we remove all numbers and inequality signs as long as our puzzle remains of the desired difficulty. However, this means that we create more difficult puzzles on average. Level 10 puzzles can often become especially difficult. All other levels are capped to some degree, but level 10 puzzles generated in this way can create unbalanced puzzles where trial-and-error is required a disproportional number of times. We solve this problem by, for example, adding a stopping chance. When generating a level 10 puzzle, each time a hint is removed after which the puzzle is considered level 10, we metaphorically flip a coin to determine whether the minimization process is continued or not. Similarly, we could specify a chance that a new hint is removed. If the puzzle is already of the desired difficulty, any time we consider a hint, we randomly choose whether to erase it or not. Notice that these solutions also speed up the generation process, as erasing fewer

hints avoids expensive calls to our solver to verify uniqueness and avoids the many computations required to determine the new difficulty.

When generating a puzzle of a certain difficulty, we can reach a puzzle state that is estimated to be of lower difficulty, but removing any hint either does not affect the rating or makes the puzzle too difficult. If such a state is encountered, we restart the entire generation process, creating a new random Latin square and removing hints. This means that generating a puzzle of a desired size and difficulty possibly requires us to discard numerous puzzles before finding a valid candidate.

## 5.4 Verifying the rating system

We conduct an experiment to study the accuracy of our rating system by comparing our rating to the rating of a different source. A popular source for Futoshiki puzzles is www.futoshiki.com, which features thousands of puzzles for each puzzle size from 4×4 to 9×9 in four difficulty categories. The lowest category is *Trivial* and consists of nearly complete Latin squares without any inequality signs. As we are interested in Futoshiki puzzles with an intermediate number of inequality signs, we do not consider this category. The other categories in increasing order of difficulty are *Easy*, *Tricky*, and *Extreme*. In this experiment, we look at three different puzzle sizes: small $5\times5$ puzzles, medium-sized $7\times7$ puzzles, and large $9\times9$ puzzles. For each puzzle size and difficulty category, we analyze 100 different puzzles from the website and assign a difficulty to each one.

Table 10: Comparison of difficulty categories on www.futoshiki.com to our difficulty rating.

| N | Difficulty | Avg. rating | Min. rating | Max. rating |
|---|---|---|---|---|
| 5 | Easy | 1.85 | 1 | 3 |
| | Tricky | 3.98 | 2 | 4 |
| | Extreme | 7.00 | 5 | 9 |
| 7 | Easy | 2.58 | 1 | 3 |
| | Tricky | 4.29 | 4 | 5 |
| | Extreme | 7.45 | 7 | 10 |
| 9 | Easy | 2.98 | 2 | 3 |
| | Tricky | 4.99 | 4 | 6 |
| | Extreme | 7.79 | 7 | 10 |

The results of the experiment are found in Table 10. We see that our average rating increases between each category for all sizes. For the most part, the categories from the website roughly correspond to our categories of *easy* (rating 1 to 3), *medium* (rating 4 to 6), and *hard* (rating 7 to 10). We see that we give smaller puzzles a lower rating in general compared to the ratings of the website. This reflects our attempt to balance the rating across puzzle sizes as all techniques are easier to apply on smaller puzzles.

This experiment shows that our rating system is on par with those of a popular Futoshiki source. To properly analyze the correctness of our rating system, we could conduct an empirical study where a large number of human puzzle solvers are given a puzzle to solve and are asked to judge its difficulty. This is outside the scope of our current research.

# 6 Advanced heuristics for solving Futoshiki puzzles

We have outlined a number of puzzle solving techniques in Section 5 and shown that these can be used to assign a difficulty rating to a Futoshiki puzzle. All of these techniques are used to eliminate candidates in cells of a puzzle. With this in mind, we reconsider our heuristic backtracking solver. This method chooses the cell with the fewest candidates in which to place a number. We examine whether using the candidate list strategy techniques in combination with the heuristic speeds up solving.

Analyzing the experiments in Section 4, we see that the heuristic backtracking algorithm shows considerable potential: it finds a solution within a franction of a second in most cases, especially in puzzles with many hints. However, in a bad case, it is significantly slower than SAT and ILP. The heuristic selects the cell with the least number of candidates. Considering that we have introduced numerous methods in Section 5.1 that are used to reduce cell candidates, we want to evaluate the effect of combining these methods with our heuristic function. To do this, before our heuristic selects a new cell to place a number in, we reduce the candidates as much as possible: we try each technique from most basic to most advanced until no technique can be applied to the puzzle state. Considering that we want this process to run in little time, we choose to include only the techniques that can be computed in polynomial time. This means that in this loop we do not use the naked multiple, swordfish, and trial-and-error techniques, as these require exponential computation time.

When backtracking, we have to be careful to go back to the precise puzzle state that we were in: we must erase all cells filled with naked and hidden singles, and we must reset the full list of candidates. With careful implementation, we avoid resetting the list of candidates when placing a number in a previously empty cell, avoiding a substantial amount of unnecessary recomputation.

Pseudocode for an iterative version of this method is shown in Algorithm 7. To measure its performance, we conduct some experiments.

## 6.1 Experiments

In this section, we compare this new solving method to the methods from Section 2. Firstly, we compare the number of states visited before encountering a solution with $\mathrm{BT_H}$ to the number of states visited with our new method, which we call $\mathrm{BT_{H+}}$, as it is a revised version of the original heuristic function. The same configurations from Section 4.1 are used, where uninteresting configurations are left out.

The results are displayed in Table 11. For configuration $(5, 10, 5)$, both methods are able to find a solution while visiting the minimum possible states on average. In all other configurations, $\mathrm{BT_{H+}}$ finds a solution earlier. The difference is especially pronounced for larger puzzles with fewer hints. This indicates that the adjustment to the heuristic function could improve performance for what previously were bad cases. To examine this, we measure the computation time required by $\mathrm{BT_{H+}}$ to find a solution to the configurations that $\mathrm{BT_H}$ performed poorly in Section 4.2.

The results are found in Table 12. The $\mathrm{BT_H}$ method did not finish computation in reasonable time for two of the configurations, and required close to a minute on average for the other two. However, $\mathrm{BT_{H+}}$ is able to find a solution in a fraction of a second on average for all four configurations. Not only is this a large improvement from $\mathrm{BT_H}$, it is also significantly faster than ILP, which we considered the best option for large puzzles. In spite of this, ILP performed relatively poorly when used to generate puzzles. To see whether $\mathrm{BT_{H+}}$ can be used for this purpose, we

**Algorithm 7** Iterative backtracking solver with advanced heuristics to find all solutions.

**Pre:** Puzzle has at least one empty cell. A global variable *solutions* is initialized with value 0.
**Post:** Variable *solutions* holds the total amount of different solutions to the puzzle.

```
reduce candidates
stack = {cell with fewest candidates}                    ▷ Ties are broken arbitrarily
while stack not empty do
    current cell = stack.pop
    if current cell not empty then                       ▷ Restore inferences made on previous state
        erase derived numbers and restore candidates
    end if
    if candidates untried in current cell then
        current cell = next candidate
        if all cells filled then
            solutions += 1
        else
            reduce candidates
            push current cell onto stack
            if puzzle valid then                         ▷ All cells have at least one candidate
                next cell = cell with fewest candidates
                push next cell onto stack
            end if
        end if
    end if
end while
```

Table 11: Comparison of the number of visited states before finding a solution between the original and the revised heuristic backtracking approaches on various puzzle configurations. The average number of visited states across 100 instances is shown.

| **N** | **Numbers** | **Inequalities** | $\mathbf{BT_H}$ | $\mathbf{BT_{H+}}$ |
|---|---|---|---|---|
| 5 | 0 | 15 | 94 | 30 |
| 5 | 10 | 5 | 15 | 15 |
| 9 | 15 | 75 | 84 911 | 162 |
| 9 | 40 | 10 | 49 | 42 |
| 12 | 50 | 150 | 63 720 | 102 |
| 12 | 75 | 25 | 174 | 72 |
| 12 | 75 | 150 | 71 | 69 |

Table 12: Measurement of the computation time to find a solution to a puzzle for the revised heuristic solving method on puzzle configurations that are considered bad cases for the original heuristic solving method. The average time across 100 instances is measured and shown in seconds, along with the standard deviation.

| **N** | **Numbers** | **Inequalities** | $\mathbf{BT_{H+}}$ |
|---|---|---|---|
| 15 | 100 | 15 | $< 0.01$ |
| 15 | 105 | 15 | $< 0.01$ |
| 25 | 400 | 25 | $0.04 \pm 0.12$ |
| 25 | 405 | 25 | $0.01 \pm 0.03$ |

measure the time required to generate 100 minimal puzzles of various sizes when using this method to verify the number of solutions to a puzzle during puzzle generation. Considering our preference for the top-down generation method, we only consider this approach.

Table 13: Measurement of the computation time required to generate minimal puzzles of different sizes using the top-down generation approach with the revised heuristic solving method. For each size, 100 puzzles are generated. The average time in seconds to generate a puzzle is shown.

| **N** | **TD** |
|---|---|
| 5 | $< 0.01$ |
| 6 | 0.01 |
| 7 | 0.12 |
| 8 | 0.98 |
| 9 | 8.52 |

The results are shown in Table 13. In Section 4.3, we saw that using SAT allowed us to generate puzzles in the least amount of time. With the TD approach, generating a minimal $7\times7$ required nearly 5 seconds on average. With $\mathrm{BT_{H+}}$ we are able to generate these puzzles in just over a tenth of a second on average, and are even able to generate $8\times8$ puzzles in just under a second on average.

These experiments show that our adjustment to the heuristic solving approach leads to a significant improvement, and it seems that this method is the most efficient of the discussed methods, not only for solving puzzles, but also for puzzle generation. It is possible that $\mathrm{BT_{H+}}$ also has a bad case, though perhaps less common or less severe than $\mathrm{BT_H}$. It could be that $\mathrm{BT_{H+}}$ is slower than other methods for certain puzzle configurations. More research is required to discover its limits.

# 7 Conclusions and further research

A Futoshiki puzzle is a Latin square completion puzzle with inequality signs between cells. In our research, we study how to generate well-designed Futoshiki puzzles. These puzzles must have a unique solution, an appropriate number of hints, and be of a suitable difficulty.

An important part of this process consists of choosing an efficient solver, as a solver is frequently required during the generation process to determine whether a puzzle has a unique solution. We

mainly consider three solving methods: a heuristic backtracking method, a method using SAT, and a method using ILP. Using SAT and ILP allow us to employ existing solvers for these problems that are highly optimized. A benefit of this is that when these solvers are improved by external sources, our performance improves with it. Although we can use existing methods, these approaches require creative thinking to carefully encode our problem as an instance of these problems.

From our experiments, we learn that the best method for solving and generating Futoshiki puzzles efficiently is an advanced heuristic backtracking method. This method chooses the cell with the fewest possible numbers, where the possible numbers are reduced at each step by several inference methods. More research is required to discover the limits of this method and whether it is the best option for all situations.

To generate puzzles, two approaches are discussed. The bottom-up approach starts from an empty square and adds numbers and inequalities. The top-down approach starts from a Latin square, adds inequalities, and removes numbers. The top-down approach allows us to more easily control the number of hints in a puzzle, and is also more time-efficient. Using this approach shifts some of the focus to creating a random Latin square as a starting point. Although we discuss methods to create Latin squares efficiently at random, we are unable to fully randomize this process without sacrificing computation time. This could be a topic of future work.

To assign a difficulty to a puzzle, we use the candidate list strategy to simulate human solving. All candidates for each cell are written down and candidates are eliminated with inference techniques until the puzzle is solved. By simulating this process, we learn which inference techniques are required to solve a puzzle. Our method focuses on finding the most advanced technique required; more research is needed to find the minimal techniques required to solve a puzzle. By ordering the techniques in terms of difficulty and assigning a score to each technique, we derive a rating from 1 to 10 based on the list of required techniques. Our rating system yields ratings similar to those from a popular source. To find out whether our ratings are accurate, an empirical study with human subjects may be required. Furthermore, changing the order of techniques in terms of difficulty or changing the scores for techniques leads to different, possibly better, results. More research is required to find the appropriate order and scores of the inference techniques for the most accurate rating.

# References


[1] Bertram Felgenhauer and Frazer Jarvis. Mathematics of Sudoku I. *Mathematical Spectrum*, 39(1):15–22, 2006.

[2] Esther Addley. If you were seduced by Sudoku, prepare for Futoshiki fever. https://www.theguardian.com/world/2006/sep/30/japan.estheraddley, September 2006. Accessed: August 10, 2026.

[3] A. Donald Keedwell and József Dénes. Classification and enumeration of Latin Squares and Latin Rectangles. In *Latin Squares and Their Applications*, chapter 4, pages 123–158. North-Holland, Boston, second edition, 2015.

[4] Banu Baklan Şen and Öznur Yaşar Diner. Ant colony optimization algorithm for the Futoshiki puzzle. *Gazi University Journal of Science*, 38(4):1754–1768, 2025.

[5] Banu Baklan Şen and Öznur Yaşar Diner. List coloring based algorithm for the Futoshiki puzzle. *An International Journal of Optimization and Control: Theories & Applications (IJOCTA)*, 14(4):294–307, October 2024.

[6] Huw Lloyd, Matthew Crossley, Mark Sinclair, and Martyn Amos. J-POP: Japanese puzzles as optimization problems. *IEEE Transactions on Games*, 14(3):391–402, 2021.

[7] Kazuya Haraguchi. The number of inequality signs in the design of Futoshiki puzzle. *Journal of Information Processing*, 21(1):26–32, 2013.

[8] Arnab Kumar Maji, Sunanda Jana, and Rajat Kumar Pal. A comprehensive Sudoku instance generator. In *Advanced Computing and Systems for Security: Volume 2*, pages 215–233. Springer, 2015.

[9] Radek Pelánek. Difficulty rating of Sudoku puzzles: An overview and evaluation. *arXiv preprint arXiv:1403.7373*, 2014.

[10] Charles J. Colbourn. The complexity of completing Partial Latin Squares. *Discrete Applied Mathematics*, 8(1):25–30, 1984.

[11] Peter van Beek. Backtracking search algorithms. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, chapter 4, pages 85–134. Elsevier, 2006.

[12] Norman Sadeh and Mark S. Fox. Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem. *Artificial Intelligence*, 86(1):1–41, 1996.

[13] Sian K. Jones, Paul A. Roach, and Stephanie Perkins. Construction of heuristics for a search-based approach to solving Sudoku. In Max Bramer, Frans Coenen, and Miltos Petridis, editors, *Research and Development in Intelligent Systems XXIV*, pages 37–49, London, 2008. Springer London.

[14] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. Association for Computing Machinery.

[15] Yakir Vizel, Georg Weissenbacher, and Sharad Malik. Boolean satisfiability solvers and their applications in model checking. *Proceedings of the IEEE*, 103(11):2021–2035, 2015.

[16] SAT Competition Organizing Committee. International SAT competition. https://satcompetition.github.io/. Accessed: April 22, 2026.

[17] Hamdy A. Taha. Integer programming. In *Mathematical Programming for Operations Researchers and Computer Scientists*, pages 41–69. CRC Press, 2020.

[18] Banu Bitgen Sungur. An integer programming formulation for the Futoshiki puzzle. *International Review of Economics and Management*, 10(2):38–49, 2022.

[19] Christos H. Papadimitriou. On the complexity of integer programming. *Journal of the ACM*, 28(4):765–768, October 1981.

[20] Manuel Eberl. Fisher–Yates shuffle. *Archive of Formal Proofs*, 2016:19, 2016.

[21] Florian Pollitt, Mathias Fleury, Katalin Fazekas, Nils Froleyks, André Schidler, Dominik Schreiber, and Armin Biere. CaDiCaL 3.0. In Alexey Ignatiev and Stefan Szeider, editors, *29th International Conference on Theory and Applications of Satisfiability Testing (SAT 2026)*, volume 377 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 40:1–40:14, Dagstuhl, Germany, 2026. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[22] Armin Biere. CaDiCaL SAT solver. https://fmv.jku.at/cadical/. Accessed: July 16, 2026.

[23] SCIP Optimization Suite Developers. SCIP optimization suite. https://scipopt.org/. Accessed: May 11, 2026.

[24] Ivo Hedtke, Joshua Morris, and SCIPpp Contributors. SCIPpp: A C++ wrapper for SCIP. https://scipopt.github.io/SCIPpp/. Accessed: May 11, 2026.

[25] Tom Davis. The mathematics of Sudoku. http://www.geometer.org/mathcircles/sudoku.pdf, September 2012. Unpublished manuscript. Accessed: June 20, 2026.